
Functionality and Dependability Assurance in Massively Networked Scenarios

Jose Ignacio Aizpurua Unanue

Supervisor:

Eñaut Muxika Olasagasti



A thesis submitted to Mondragon Unibertsitatea
for the degree of Doctor of Philosophy

Department of Electronics and Computer Science
Mondragon Goi Eskola Politeknikoa
Mondragon Unibertsitatea

October 2014

“Beti izango haiz gure bihotzetan, adiorik ez Bene, gero arte baizik.”

In the memory of Gorka Ramos Hernández (04-07-2012).

Abstract

The design of dependable systems and reduction of economic costs have been viewed as conflicting goals. Traditional dependable design approaches replicate system resources to improve fault tolerance. However, the aggregation of hardware, software or communication resources to add recovery capabilities to a system function results in higher costs.

Instead of adding redundancies that provide recovery capabilities to a predefined system function, in Massively Networked Scenarios (MNS) there is room to take advantage of over-dimensioning design decisions and overlapping structural functions by using heterogeneous redundancies: components that, besides performing their primary intended design function, restore compatible functionalities of other components. MNS are systems characterized by several replicas of system functions distributed throughout the physical structure (e.g., a train has replicated functions throughout its cars; or buildings have replicated functions throughout its floors and rooms). Besides, in these scenarios there are many processing units, sensors and actuators connected to a communication network for different purposes.

We have designed a methodology named D3H2 (aDaptive Dependable Design for systems with Homogeneous and Heterogeneous redundancies) to design HW/SW architectures systematically applying modelling and analysis approaches. These approaches include the systematization of the next activities: (1) identification of heterogeneous redundancies; (2) integration of redundancies in the HW/SW architecture including necessary fault detection, reconfiguration and communication implementations; and (3) dependability and cost assessment of the designed HW/SW architectures.

Through the application of the devised modelling and analysis approaches, D3H2 enables the assessment of the effect of alternative redundancy and reconfiguration strategies, fault detection and communication implementations on system dependability and cost. The methodology has been applied to non-repairable and repairable systems.

Design strategies based on heterogeneous redundancies have shown potential to improve system dependability cost-effectively. However, the de-

cision of which redundancy strategy is better for a specific system function should be evaluated case-by-case basis through the application of the D3H2 methodology.

An experimental prototype using real railway communication elements has been developed to validate some of the concepts treated in the D3H2 methodology.

Resumen

El diseño de sistemas confiables y la reducción de costos han sido vistos como objetivos conflictivos. Las técnicas tradicionales para diseñar sistemas confiables replican los recursos del sistema para mejorar la tolerancia a fallos. Sin embargo, añadir recursos de hardware, software o de comunicaciones para proporcionar capacidad de recuperación al sistema resultan en un incremento de costes.

En vez de añadir redundancias que proporcionan capacidad de recuperación a una función predefinida en los Escenarios Masivamente Redundados (EMR) hay opción para aprovechar las decisiones de diseño sobre-dimensionadas y funciones que se solapan usando redundancias heterogéneas: componentes que además de desarrollar su función principal, pueden recuperar las funcionalidades compatibles de otros componentes. Los EMR son sistemas caracterizados con varias replicas de las funciones del sistema distribuidos en toda su estructura física (p.e., un tren tiene funciones replicadas en sus coches; o los edificios tienen funciones replicadas en diferentes plantas y habitaciones). Además, en estos escenarios hay varias unidades de procesamiento, sensores y actuadores conectados a una red de comunicaciones con diferentes objetivos.

Hemos diseñado la metodología D3H2 (aDaptive Dependable Design for systems with Homogeneous and Heterogeneous redundancies) para diseñar arquitecturas HW/SW sistemáticamente aplicando técnicas de modelado y análisis. Estas técnicas incluyen la sistematización de las siguientes actividades: (1) identificación de las redundancias heterogéneas; (2) integración de las redundancias en las arquitecturas HW/SW incluyendo las implementaciones de detección de fallos, reconfiguración y comunicación; y (3) evaluación de la confiabilidad y costo de las arquitecturas HW/SW diseñadas.

Mediante la aplicación de las técnicas diseñadas de modelado y análisis, D3H2 permite la evaluación del efecto de las estrategias alternativas de redundancia y reconfiguración, y de las implementaciones de detección de fallos y comunicación en la confiabilidad y el costo del sistema. La metodología ha sido aplicada tanto a sistemas reparables como no reparables.

Las estrategias de diseño basadas en redundancias heterogéneas han demostrado potencial para mejorar la confiabilidad del sistema sin comprometer el costo. Sin embargo, la decisión de qué estrategia de redundancias es mejor para una función específica debe ser evaluado uno por uno mediante la aplicación de la metodología D3H2.

Para evaluar algunos conceptos desarrollados en la metodología D3H2 se ha desarrollado un prototipo experimental usando elementos de comunicaciones reales de la industria ferroviaria.

Laburpena

Sistema fidagarrien diseinua eta kostu ekonomikoaren murrizketa helburu bateraezin bezela kontsideratu izan ohi dira. Diseinu teknika tradizionalak sistemako errekurtsuak bikoiztu izan ohi dituzte akatsekiko tolerantzia hobetzeko. Hala ere, hardware, software eta komunikazio errekurtsuak gehitzeak sistemaren kostua igotzea dakar.

Erreduantzia esplizituak gehitu beharrean aukeratutako funtzioei errekupeazio gaitasuna emateko, Masiboki Saretutako Eszenategietan (MSE) posible da gain-dimentsionatutako diseinu erabakiak eta errepikatutako funtzioak aprobetxatzea erreduantzia heterogeneoak erabiliz: hauek beraien diseinuko helburu nagusia betetzeaz gain beste osagaien funtzio bateragarriak errekupeatzeko gai da. MSE-ak sistemako funtzioen hainbat kopiaz osatuta daude eta hauek sisteam osoan zehar banatuta daude (adibidez, tren batek bere kotxeetan errepikatutako funtzioak ditu; edo eraikuntzek errepikatutako funtzioak dituzte beraien solairu eta gelatan zehar). Gainera, eszenatoki horietan hainbat prozesamendu unitate, sentore eta eragile daude komunikazio sarera konektatuta helburu ezberdinekin.

D3H2 (aDaptive Dependable Design for systems with Homogeneous and Heterogeneous redundancies) metodologia diseinatu dugu HW/SW egiturak sistematikoki diseinatzeko erduztapen eta analisi teknikak erabiliz. Teknika hauek, ondoko jardueren sistematizazioa egiten dute: (1) erreduantzia heterogeneoen identifikazioa; (2) erreduantzia heterogeneoen txertatzea HW/SW egituran beharrezko akats detekzio, birkonfigurazio eta komunikazio inplementazioak gehituz; eta (3) diseinatutako HW/SW egituraren fidagarritasun eta kostu analisia.

Diseinatutako erduztapen eta analisi teknikak aplikatuz, erreduantzia eta birkonfigurazio estrategia ezberdinen, akats detektatzaile eta komunikazio inplementazioen eragina aztertzen du D3H2k sistemaren fidagarritasun eta kostuan. Metodologia sistema konpongari eta konponezinei aplikatu zaie.

Erreduantzia heterogeneotan oinarritutako diseinu estrategiak sistemaren fidagarritasuna kostua konpromezuan jarri gabe hobetu dezake.

Hala ere, zein erredundantzia mota den hobea funtzio espezifiko bakoitzarentzat kasuz kasu aztertu beharrekoa da D3H2 metodologiaren bidez.

D3H2n garatutako kontzeptu batzuk baieztatzeko prototipo esperimental bat garatu da benetako trenen komunikazio elementuak erabiliz.

Eskertza

First of all I would like to thank to my thesis director, Eñaut Muxika, for the support, assistance and guidance that he gave me during the Ph.D. Eñaut gave me the freedom to do whatever I wanted, at the same time continuing to contribute valuable feedback and encouragement. I also would like to thank to Mondragon University and CAF Power & Automation for funding and giving me the opportunity to develop this work.

I gratefully acknowledge the members of my Ph.D. committee for their time to read and review this dissertation.

I am also grateful to my Ph.D. student colleagues: Aritz, Enaitz, Alain, Aitor L., Aitor A., Dani, Oscar and Raul; as well as to those ex-Ph.D. students colleagues who finished their Ph.D. successfully during these years: Iker S., Maitane, Iker Z., Peio, Lorena, Lorea, Idoia, Iñaki and Maite. I wish to thank also to my colleagues in the CAF Power & Automation laboratory at Mondragon University for creating a nice working environment: Ane, Egoitz, Mikel and Unai.

Besides, I would like to thank Iokin Azkue and Juan Fernández for making possible my stay at CAP Power & Automation and introducing to the industry. I am also grateful to my colleagues there: Gari, Jon, Esti, Juanmi and Oihana for making my stay unforgettable.

I wish to thank also my Italian colleagues Gabriele and Ferdinando, who have provided me support and guidance in those moments that I needed. Your help pushed me to improve myself and stimulate my passion for reliability & safety engineering.

I am specially thankful to Yiannis Papadopoulos for making possible my stay at the University of Hull and making me feel as if I were at home. Your wisdom and fruitful comments are always inspirational. Thanks also to my colleagues in the Dependable Systems Research Group for our interesting discussions: Leonardo, Martin, David, Sohag, Luis, and Sheptavera.

And last but not least, I would like to express my deepest gratitude to my family and friends. To my dear parents and to my exceptional cousins, Josune and Alazne, who encouraged me to not throw in the towel and specially to Paula, for her unwavering support and encouragement during these years.

This work would not have been possible without your support and encouragement. Cheers!

Contents

Contents	xi
1 Introduction	1
1.1 Opportunity Identification	1
1.2 Scope of the Research	3
1.3 Research Objectives	5
1.4 Research Hypothesis, Contributions & Limitations	6
1.5 Research Methodology	7
1.6 Thesis Outline	9
2 Literature Review	11
2.1 Application Framework	11
2.2 Dependability Framework	18
2.2.1 Dependability: Definitions and Classifications	18
2.2.2 Designing for Fault Tolerance and Dependability	24
2.2.3 Fault Hypothesis & Failure/Error Model	28
2.2.4 Opportunity Analysis	30
2.3 Overview of the Main Dependability Analysis Approaches	32
2.3.1 Hybrid Approaches	33
2.3.2 Opportunity Analysis	47
2.4 Design of Dependable Systems: Trade-Off Between Dependability & Cost	48
2.4.1 Design Approaches using Homogeneous Redundancies	49
2.4.2 Design Approaches using Heterogeneous Redundancies	52
2.4.3 Opportunity Analysis	60
2.5 Conclusions	64
3 D3H2 Methodology	65

3.1	Introduction	65
3.2	Overview of the D3H2 Methodology	66
3.3	HW/SW Architecture Design	68
3.3.1	Functional Modelling Approach	69
3.3.2	Compatibility Analysis	73
3.3.3	Reconfiguration Strategies	76
3.3.4	Extended Functional Modelling Approach	77
3.4	Results	81
3.5	Conclusions	96
4	Dependability & Cost Analysis of Non-Repairable Systems	99
4.1	Introduction	100
4.2	Dependability Evaluation Modelling Approach	102
4.2.1	Concepts and Notation	102
4.2.2	Analysis Algorithm	104
4.2.3	Analysis of the State of the Art Approaches	108
4.2.4	Implementation: Component Dynamic Fault Trees	109
4.3	Sensitivity Analysis	114
4.3.1	Simulation-based Importance Measurement Indices	114
4.3.2	Implementation of the Sensitivity Analysis	116
4.4	Uncertainty Analysis	119
4.5	Cost Analysis	122
4.6	Results	123
4.6.1	Fire Protection Control	123
4.6.2	Door Status Control	133
4.7	Conclusions	145
5	Dependability & Cost Analysis of Repairable Systems	149
5.1	Introduction	149
5.2	Dependability Evaluation Modelling Approach for Repairable Systems	152
5.2.1	Concepts and Notation	152
5.2.2	Analysis Algorithm	154
5.2.3	Implementation	157
5.3	Cost Analysis	160
5.4	Results	161

5.4.1	SAN Generic Models	161
5.4.2	Fire Protection Control	182
5.4.3	Door Status Control	188
5.5	Conclusions	193
6	D3H2 Methodology: Experimental Evaluation	197
6.1	Introduction	197
6.2	Industrial Railway Communication Architectures	198
6.2.1	Communication Networks	198
6.2.2	Communication Devices	199
6.3	Application Architecture	203
6.3.1	Scenario I: Sensor-Level Reconfiguration	205
6.3.2	Scenario II: PU-Level Reconfiguration	207
6.3.3	Scenario III: Communication-Level Reconfiguration	208
6.4	Conclusions	209
7	Conclusions and Future Work	211
7.1	Conclusions	211
7.2	Contributions	216
7.3	Future Work	217
	Appendices	219
A	Overview of the Basic Dependability Analysis Approaches	221
A.1	Event-Based (Combinatorial, Static) Approaches	221
A.2	State-Based (Dynamic) Approaches	227
B	Classification of the Hybrid Approaches and Tool Support	237
B.1	Classification of the Hybrid Approaches	237
B.2	Tool Support	238
C	Analysis of Literature Approaches on a System Example	241
C.1	(Static) Fault Tree [Vesely02]	242
C.2	Component Fault Tree (ESSaReL tool) [Kaiser03]	243
C.3	HiP-HOPS [Papadopoulos11]	244
C.4	Repairable Dynamic Fault Tree (RAATSS tool) [Manno14c]	245
C.5	Structure Function of Dynamic Fault Trees [Merle14]	246
C.6	BDMP [Bouissou07]	247

C.7	SEFT - DSPN [Kaiser07]	248
D	Automation/Implementation of the HW/SW Architecture Design	253
D.1	Annotations of the System Architecture	254
D.2	Identification of Heterogeneous Redundancies	256
D.3	Extraction of the Reconfiguration Table	259
E	Failure Rate & Cost Data	263
F	PAND Model for Repairable Systems	267
	List of Figures	275
	List of Tables	279
	List of Algorithms	281
	Glossary	284
	Bibliography	285
	List of Abbreviations	311

Introduction

This chapter describes the motivation that inspired the author to research in the field of model-based reliability engineering so as to provide solutions to the examined problems. The chapter is organised as follows:

- Section 1.1 describes the main motivation of this thesis.
- Section 1.2 frames the scope of this research.
- Section 1.3 defines the research objectives of this dissertation.
- Section 1.4 sets the research hypothesis, contributions and limitations.
- Section 1.5 explains the followed methodology to obtain the research objectives.
- Section 1.6 describes the structure of this thesis.

1.1 Opportunity Identification

The design of dependable systems and reduction of economic costs have been viewed as conflicting goals (e.g., see [[Somani97](#); [Elegbede03](#); [Izosimov05](#)]). Traditional dependable design approaches aim at replicating resources in order to improve fault tolerance. For instance, the widely adopted Triple Modular Redundancy (TMR) [[Avizienis85](#)] (cf. Figure 2.15) is one example among many other fault tolerance strategies that explicitly add software and/or hardware components (either same or diverse) in order to improve system dependability [[Laprie92](#); [Laprie95](#)].

Nevertheless, the aggregation of resources leads to more failure sources and higher costs. Therefore, one feasible direction to construct dependable systems and reduce the eco-

conomic cost is the optimization of system resources. To do so, we focus on the design of distributed Networked Control Systems (NCSs) [Wang08].

In distributed NCSs, remote sensors, control algorithms allocated at Processing Units (PUs), and actuators work in cooperation to perform a system function. The underlying characteristics of distributed NCSs (distributed nature, computing capacity of the networked PUs) make NCSs suitable to adapt their behaviour in the presence of system changes such as component failures or attachment of new devices.

Traditionally sensors and actuators perform a single function, while PUs handle multiple tasks. For instance, consider the air conditioning control and fire protection control functions implemented in a room: for the air conditioning control a temperature sensor measures the temperature of the room and a heater warms the room accordingly; while in the case of fire protection control a smoke detector detects the presence of smoke and a sprinkler extinguishes the fire of the room. Despite being independent control functions, it is not strange to allocate both control functions - air conditioning control and fire protection control - in the same PU.

In this work, we concentrate on optimising system resources to reduce system cost and improve the dependability of system functions. To this end, the functionalities of sensors and actuators are extended beyond their nominal design functions so as to perform as many functions as possible and feasible. Retaking the previous example of the air conditioning control and fire protection control functions and assuming that there exists another room next to the previously described one with the same functionalities, it is possible to reuse: (1) the temperature sensor to approximate the temperature of the contiguous room or (2) the sprinkler (either in the same or in the contiguous room) to raise an alarm when speakers are not working. All the hardware resources including sensors, actuators and PUs, which are able to perform additional functions beside their nominal design functions are named *heterogeneous redundancies* [Aizpurua12a] (see Section 2.2.2).

Unfortunately, the use of *heterogeneous redundancies* is not a panacea. Although the employment of heterogeneous redundancies may reduce the hardware cost and improve the dependability level of a system design without redundancies, it also introduces some drawbacks. When making use of system resources in further circumstances beside from

their nominal design consideration, additional costs emerge. Namely, it necessary to:

1. Identify and evaluate the potential resources which could provide additional compatible functionalities without incurring a considerable extra cost (i.e., identify reusable resources).
2. Adjust the system architecture with health management functionalities and implementations (i.e., fault detection and reconfiguration) to make the use of heterogeneous redundancies in further system contexts possible.
3. Evaluate the dependability and cost of the resulting system architecture.

Given the methodology to address these issues, there is room in NCSs and more specifically in massively networked scenarios (cf. Section 1.2) to optimize the use of system resources by means of *heterogeneous redundancies*.

In the literature there exist many approaches focusing on the adaptation of the system architectures to deal with component failures, however, those which address the utilization of heterogeneous redundancies or similar concepts are not many. Interestingly, when encompassing the system design process as a whole accounting for dependability, adaptivity, and heterogeneous redundancy-like issues, existing solutions are scarce (cf. Chapter 2). Therefore, the contribution of this thesis proposes the generation of a design methodology in order to evaluate the dependability and cost level of alternative architectures which make use of heterogeneous and/or homogeneous redundancies.

1.2 Scope of the Research

In order to set the framework of this thesis and define the scope of this research we define the application context in which this work is situated and later, we will define the faults that the proposed approach is intended to deal with (see Chapter 2).

Massively Networked Scenarios

The application context of this dissertation is framed within NCSs operating in massively networked scenarios: systems characterized by several replicas of system functions

distributed throughout the physical structure. In these scenarios there are many PUs, sensors and actuators connected to a communication network for different purposes.

For instance, as Figure 1.1 depicts, a train is an example of a NCS operating in massively networked scenarios. A train has replicated functions throughout its cars, each car has implemented its own functions and (some of) these functions are replicated throughout the different cars of the train.

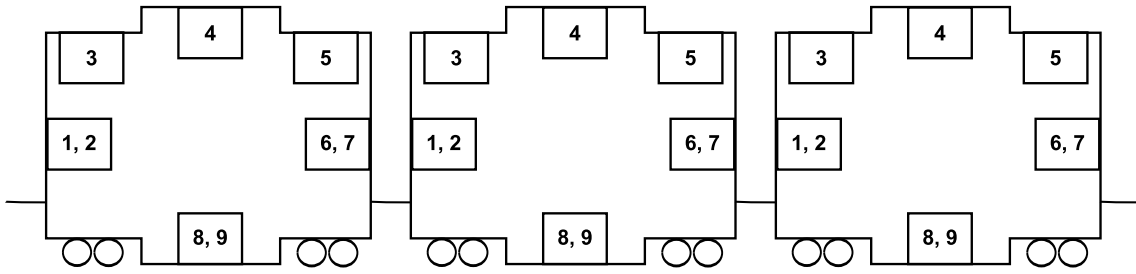


Figure 1.1: Massively Networked Scenario: Railway Train Example

The functions¹ numbered in Figure 1.1 are defined as follows:

1. Air Conditioning Control
2. Passenger Alarm System
3. Fire Protection Control
4. Video Surveillance
5. Intelligent Light Control
6. Passenger Information System
7. Voice Communication Control
8. Door Status Control
9. Passenger Counting System

Each function enclosed within a rectangle has its own set of resources (i.e., sensors, controllers and actuators). For further details about the implementation of these functions please refer to Section 2.1.

¹There exist other functions which have not been represented here, e.g., braking control or power control.

Another example of NCSs operating in massively networked scenarios are the buildings. Buildings are constituted by floors and rooms, which have replicated functions throughout its floors and rooms as Figure 1.2 shows.

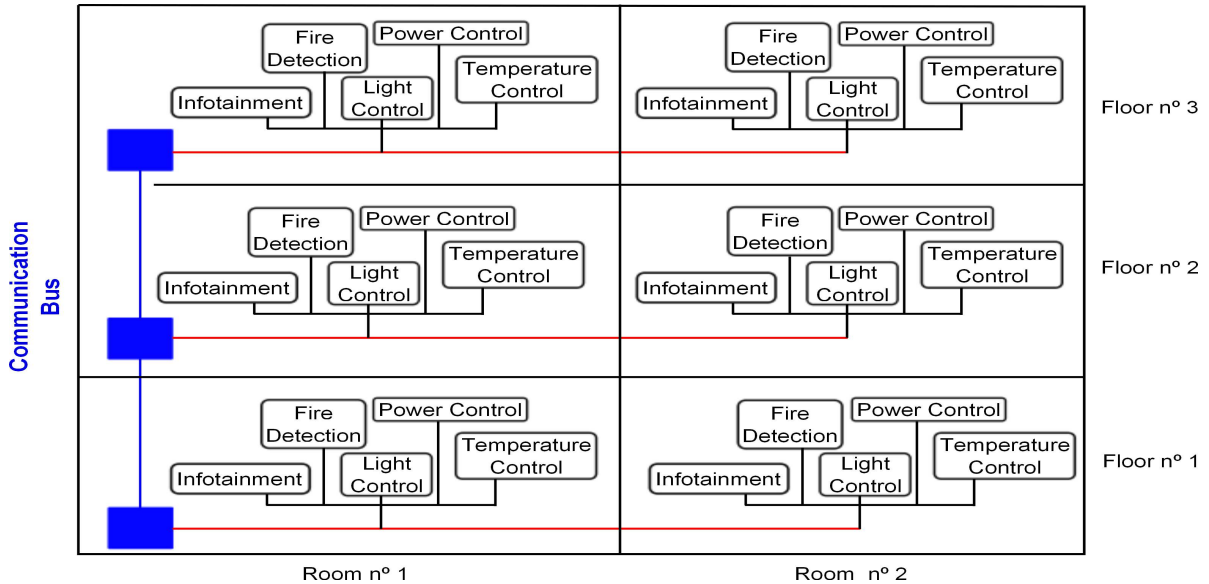


Figure 1.2: Internal Architecture of a Building: Functions and Communication Interfaces

Therefore, we concentrate on studying NCSs operating in massively networked scenarios so as to exploit the potential heterogeneous redundancies which may exist in these systems.

1.3 Research Objectives

The main goal of this thesis is to **evaluate the impact of the reuse of system resources on the overall system dependability and analyse whether it reduces the system development cost**. Starting from this main goal and after performing the study of the state of the art (cf. Section 2), further underlying necessary objectives have been defined.

Therefore, the main research goal is divided into the following research objectives of this dissertation:

1. Systematic identification of heterogeneous redundancies.

2. Systematic characterization of HW/SW architectures fitted with health management functions and their implementations, i.e., fault detection and reconfiguration.
3. Systematic evaluation of the influence of the type and number of redundancy and reconfiguration strategies on system dependability and cost.
4. Optimization of the design of control system architectures in massively networked scenarios, maximizing dependability and minimizing the cost.
5. Definition of a guideline to decide when the reuse of system resources is beneficial for the system (reduce costs, improve dependability) and when it is better to use homogeneous redundancies.

1.4 Research Hypothesis, Contributions & Limitations

The objective of this section is to define the research hypothesis in order to specify the foundations of this dissertation as well as the contributions and limitations.

As a result of the performed literature study - in which we review the works related with this thesis (cf. Chapter 2) - and linking this survey with our research objectives (cf. Section 1.3), the research hypothesis that we are going to work with is defined as follows:

*“The **systematic consideration** of the **effect of homogeneous and heterogeneous redundancies, fault detection, reconfiguration and communication functions**, allows the **optimization** of control system architectures in massively networked scenarios, **maximizing dependability and minimizing the cost.**”*

Comparing this work with already existing approaches which make use of heterogeneous redundancies for designing adaptive dependable systems, this dissertation differs in the following aspects - contributions:

- The systematization of the identification of heterogeneous redundancies.

- The explicit consideration of the faulty behaviour of the fault detection, reconfiguration and communication implementations when addressing heterogeneous redundancies.
- The systematic characterization of HW/SW architectures fitted alternative redundancy and reconfiguration strategies, fault detection implementations and reconfiguration resources.
- The systematic evaluation of the effect on dependability and cost of the designed HW/SW architectures by considering both non-repairable and repairable resources.

The following parts were left out of the scope of this work - limitations:

- The process for obtaining the failure rate data of software resources is not considered and it is assumed a known data. Nevertheless, to deal with uncertain data we have implemented an uncertainty analysis approach in Chapter 4.
- Exact solutions are not obtained, instead we concentrate on simulation techniques.
- Analysis of low-level requirements: timing requirements that components should meet in order to be compatible and perform a function timely; memory and processing power requirements that the processing units should meet; and bandwidth constraints of communication protocols that the system have to adhere will not be addressed.

1.5 Research Methodology

The proposed methodology allows the validation of the stated research hypothesis. Our research methodology is based on the characterization (design) of modelling and analysis activities and their combined application to theoretical case studies in order to validate the research hypothesis.

To this end, we have divided the research problem into smaller problems and define solutions to each of them, so that evidence is shown for each part in particular and for the overall problem in general:

- **Systematic consideration** of the implied attributes/variables in the research hypothesis - characterization of the **design model**:
 - **Definition** of a **generic system model** to design HW/SW architectures systematically including the next activities:
 - * **Systematic** identification of **heterogeneous redundancies**.
 - * **Procedural** consideration of **fault detection, reconfiguration, and communication functions**.
- **Systematic evaluation** of the effect of the implied attributes/variables in the research hypothesis on dependability and cost - characterization of the **analysis model**:
 - **Definition of analysis models and algorithms** to **evaluate systematically** the **dependability** and **cost** of the alternative HW/SW architectures designed with the **generic system model**:
 - * Definition of the **dependability** metric/model and an algorithm to analyse the dependability level of alternative HW/SW architectures systematically.
 - * Definition of the **cost** metric/model and an algorithm to analyse the cost of alternative HW/SW architectures systematically.
 - * **Overall evaluation** of the system's dependability and cost and **trade-off analysis** between these attributes.
- **Automation** of all the previous phases.
- **Validate** the feasibility of the proposed approach by using real hardware, software, and communication elements.

All these activities have been validated case by case basis through the development of theoretical case studies. Besides, to validate the feasibility of the proposed methodology, a real proof-of-concept has been developed using real hardware, software and communication elements of the railway industry² (cf. Chapter 6).

²The author was a visiting researcher at CAF Power and Automation (www.cafpower.com) for the last six months of 2013.

1.6 Thesis Outline

This report is divided into 7 chapters. The following points overview the organisation of each chapter:

- Chapter 2 defines the **application example** used to illustrate the concepts emerging from this dissertation and it provides the needed **background literature** for the development of this thesis. Through an exhaustive literature analysis of model-based system engineering and reliability engineering fields, the **opportunity (motivation)** is identified.
- Chapter 3 describes the main contribution of this dissertation: **the aDaptive Dependable Design for systems with Homogeneous and Heterogeneous redundancies (D3H2) methodology**. The methodology integrates all the research objectives and activities identified previously. For the sake of clarity, the explanation of the methodology is divided in two parts:
 1. Modelling and analysis activities to create an extended HW/SW architecture.
 2. Dependability and cost analysis of the extended HW/SW architecture.

This chapter overviews the main activities of the methodology and describes the steps to create an extended HW/SW architecture.

- Taking the extended HW/SW architecture as a starting point, Chapter 4 defines the **dependability evaluation** algorithm for **non-repairable systems** and its implementation by using simulation techniques.
- Taking the extended HW/SW architecture as a starting point, Chapter 5 defines the **dependability evaluation** algorithm for **repairable systems** and its implementation.
- Chapter 6 describes the **implementation** of the research objectives by using real railway hardware and communication elements.
- Chapter 7 sets the **conclusions** of this thesis and future research goals.

Literature Review

In this chapter we provide the necessary background information for the elaboration of the thesis and we review the previous related work so as to support our research hypothesis and set the topic of this dissertation. Besides, in order to have a consistent ongoing example throughout the thesis, we will also specify the example case study so that we can illustrate the emerging concepts directly.

The chapter is organised as follows:

- Section 2.1 describes the illustrative application framework.
- Section 2.2 presents the dependability framework for the development of this thesis introducing relevant definitions and concepts.
- Section 2.3 classifies and examines the main dependability analysis techniques.
- Section 2.4 reviews the scientific literature examining those approaches which design adaptive dependable systems by using *homogeneous* and/or *heterogeneous redundancies*.
- Section 2.5 concludes this chapter with conclusions that will determine the orientation of the forthcoming chapters of this thesis.

2.1 Application Framework

The goal of this section is to introduce the *running example* so that all the examples throughout this dissertation have a unique consistent reference. The illustrative case study has been inspired from the direct application of this thesis: a train operating in massively networked scenarios. A train (usually) is constituted by more than one car,

and each car in turn has a set of different functions, which are replicated for each car of the train.

Figure 2.1 depicts some of the functions performed in a train car that will be used throughout this thesis for illustration purposes. Different functions are connected to different communication networks and there is an interconnecting gateway element, which makes possible the communication of resources connected to different communication networks. In order to make the communication between different cars of the train possible, train switches are used. The inter-car communication and intra-car communication are implemented according to the IEC 61375 (Train Communication Network) standard [IEC07].

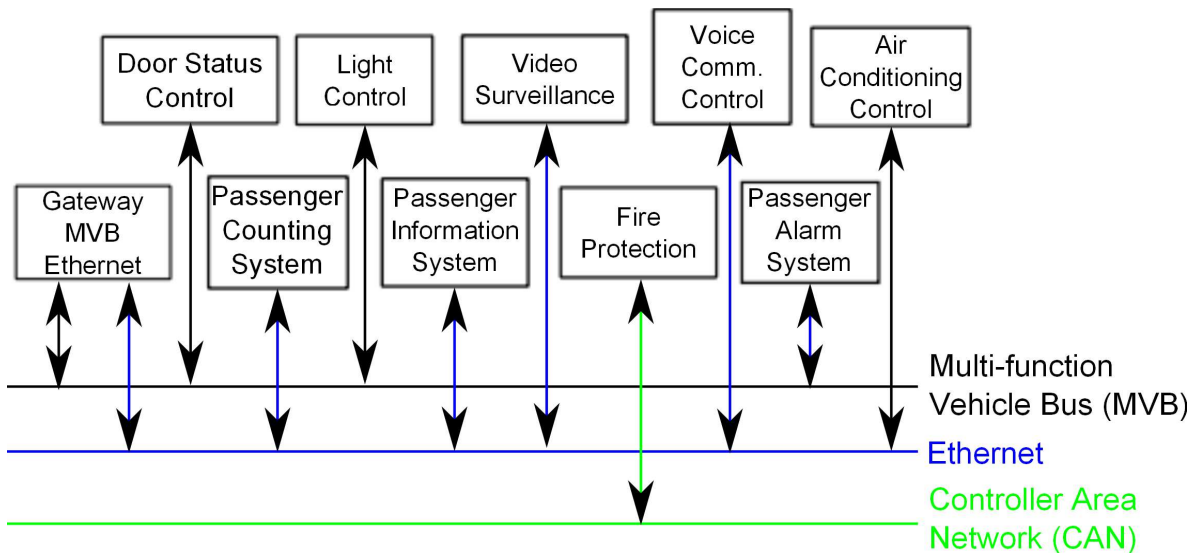


Figure 2.1: Train Car Configuration: Functions and Communication Interfaces

In Figure 2.1, each function is enclosed in a box and in turn, they have their own hardware and software resources. Consecutively we will explain the main HW/SW features of the functions and they will be used throughout this dissertation to perform different analyses.

Without loss of generality, henceforth we will assume that each car of the train will have 2 compartments ($Zone_A$, $Zone_B$) and in each compartment there will be 2 doors located side by side (cf. Figure 2.2).

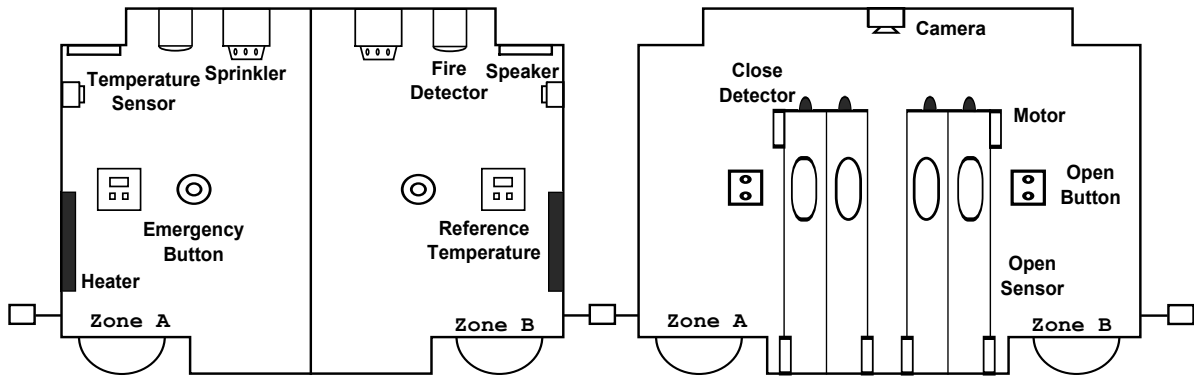


Figure 2.2: Train Car Configuration: Physical Distribution

Door Status Control

Each door in the train has many sensors and control buttons for the passengers and the driver. The doors closure is controlled by the driver based on an enable signal that will be received depending on the status of the train, e.g., while the train is running the doors must remain closed.

In the train there is a component called Train Control Monitoring System (TCMS), which controls and monitors several critical systems of the train such as traction and doors. This component is homogeneously duplicated in two reliable PUs (PU_{TCMS}) for safety purposes. The TCMS receives information about the speed of the train and it will not allow the driver to open the doors while the train is running. To this end, it provides an enable to the driver to inform about the safe operation of door opening or closing (known as *Enable Door Driver*). The driver accordingly provides an enable to the controller of each door (known as *Enable Door Passenger*) to act safely on opening or closing the doors while taking into account if the train is moving and if there is an obstacle in the door.

As Figure 2.3 shows, in a train car there is one opening and closing button for the driver connected to the driver's PU (PU_{driver}), while each door has: one opening button for passengers, one door velocity sensor, one door open detection sensor, one door closed detection sensor and one obstacle detection sensor. All these sensors, their controllers and the door control algorithm are located in the PU_{Door} .

Figure 2.4 depicts the control loop including the physical system (*Train Car Door*).

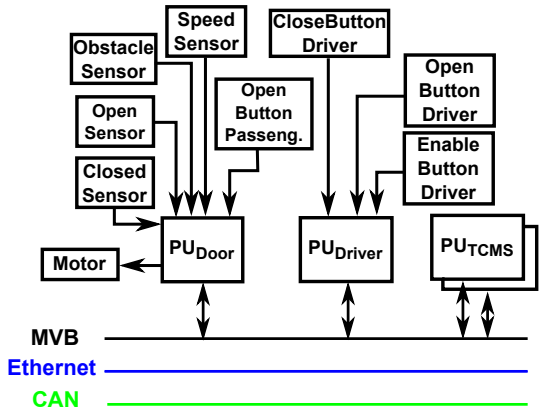


Figure 2.3: Hardware Model of the Door Status Control Function

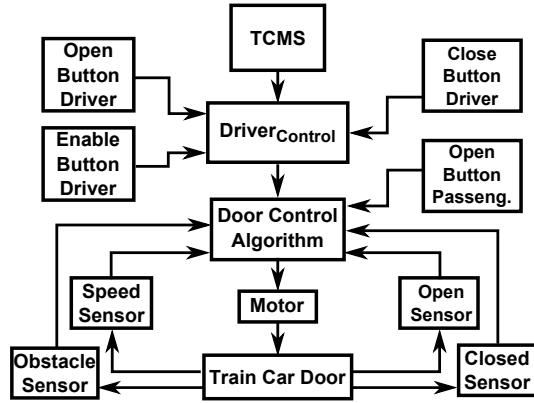


Figure 2.4: SW/Dependency Model of the Door Status Control Function

The *DriverControl* software resource located at PU_{Driver} receives the status information of the train from the TCMS component (*Enable Door Driver*) and based on the received information and driver's open, close or enable indications, it provides the *Enable Door Passenger* signal to the *Door Control Algorithm*. *Door Control Algorithm* located at PU_{Door} receives the status data of the sensors of the door and passenger commands. Then, based on driver's enable command (*Enable Door Passenger*) it will actuate on the corresponding motor to open or close the door of the corresponding compartment of the train car.

Video Surveillance

The Video Surveillance function performs monitoring tasks on each car of the train. Each car is equipped with a camera which focuses towards the doors in order to prevent hazards and injuries.

The incoming images recorded by the camera are processed through an image processing algorithm (*Process Image*) located in PU_{Cam} and in the presence of a hazard in any of the cars it raises an emergency signal using the lamps and the siren. Besides, for security issues, all the images are stored in a server connected to the same Ethernet communication network.

As depicted in Figure 2.6, the *Process Image* algorithm located in PU_{Cam} evaluates

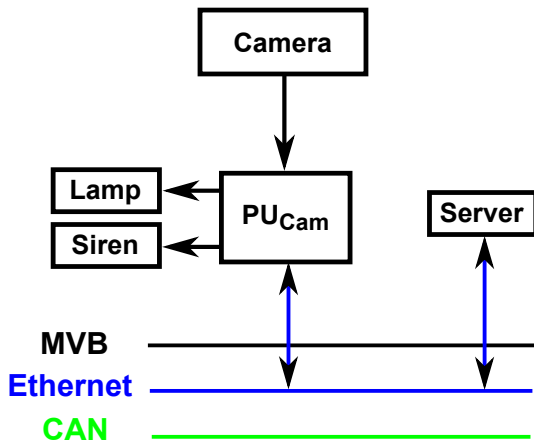


Figure 2.5: Hardware Model of the Video Surveillance Function

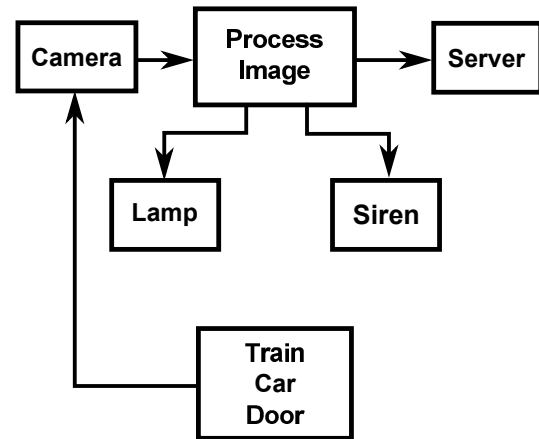


Figure 2.6: SW/Dependency Model of the Video Surveillance Function

hazardous situations in a train car; redirects the camera images towards the storage server; and raises the alarms when hazardous situations are detected.

Air Conditioning Control

The Air Conditioning Control (ACC) sets the temperature of a train car according to the reference temperature defined by the driver.

The driver is responsible for (1) activating the Air Conditioning Control on the car(s) that he/she decides - *Activate ACC*; and (2) set the reference temperature of the corresponding car. In each train car's compartment, there are dedicated PUs to perform the Air Conditioning Control of the car (PU_{ACC}). To this end, in each car the PU_{ACC} receives the current temperature of the car through a temperature sensor and heats the room by using a dedicated heater. Normally, a train car comprises of different compartments and accordingly, there exist a temperature sensor and a heater for each compartment of the train car (cf. Figure 2.7).

As depicted in Figure 2.8, each train car compartment has its own control loop so as to heat the room according to the reference temperature set by the driver.

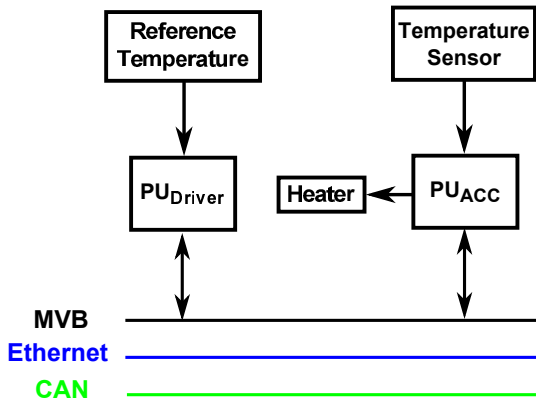


Figure 2.7: Hardware Model of Air Condition Control Function

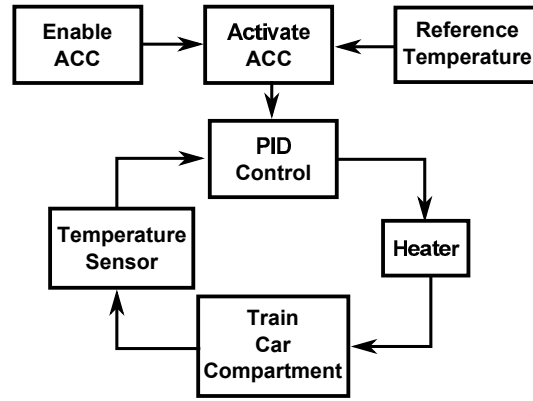


Figure 2.8: SW/Dependency Model of Air Condition Control Function

Fire Protection Control

The Fire Protection (FP) control function aims at the fire detection and extinction in a train car compartment. The hazardous situation is detected by a fire detector, which based on the presence of smoke raises a signal, or it is triggered by a user who pushes the emergency button to indicate an emergency situation (cf. Figure 2.9).

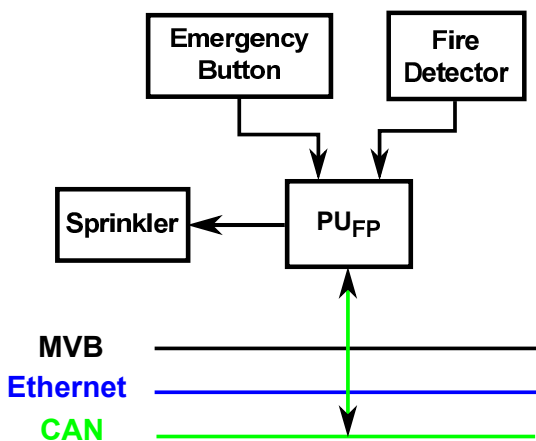


Figure 2.9: Hardware Model of the Fire Protection Function

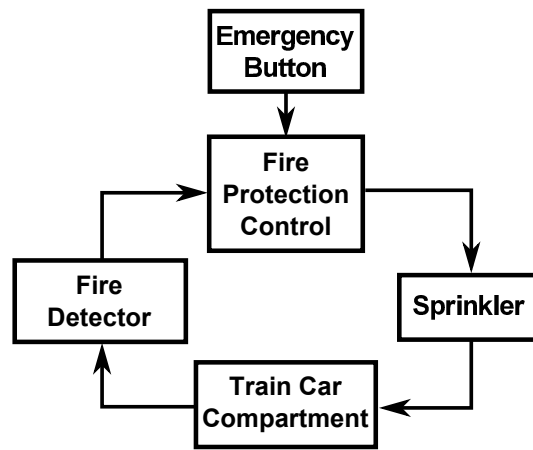


Figure 2.10: SW/Dependency Model of the Fire Protection Function

As depicted in Figure 2.10, each train car compartment has its own Fire Protection control loop so as to extinguish the possible fires. The *Fire Protection Control* SW element located in PU_{FP} activates sprinklers which are strategically located in each

compartment of the train cars.

Passenger Information System

The Passenger Information System (PIS) informs the passenger about the position of the train and next stops (cf. Figure 2.11).

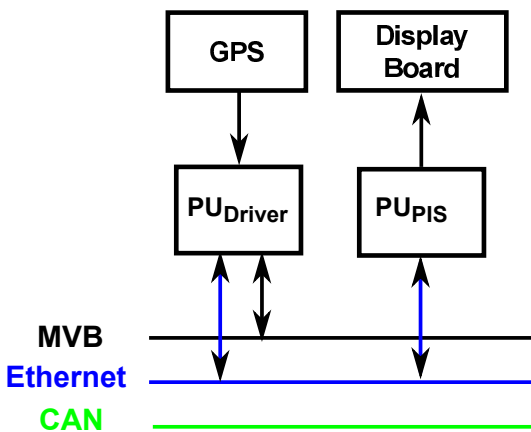


Figure 2.11: Hardware Model of the Passenger Information System

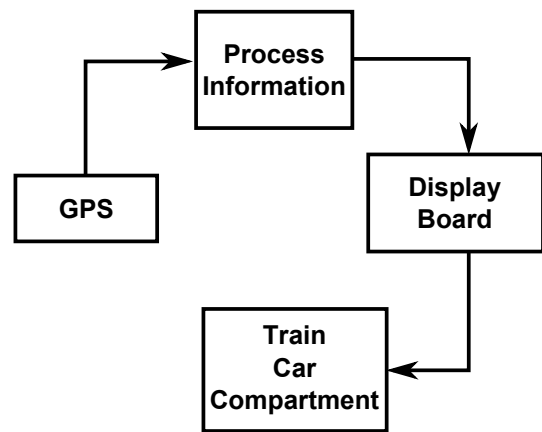


Figure 2.12: SW/Dependency Model of the Passenger Information System

The PIS function makes use of the Global Positioning System (GPS) device located at driver's compartment in PU_{Driver} . Based on the location of the train, information displays are updated with the corresponding information at runtime through the *Process Information* SW algorithm located at PU_{Driver} .

Light Control

Each car of the train may have an intelligent lighting control system, which switches on/off the lights or lowers the light intensity automatically based on the presence/absence of people. To this end, there is a presence sensor which detects if anyone is in a train car and besides, the driver has its own manual activation button for those cases in which the sensor is not working correctly (cf. Figure 2.13).

The *Light Control Algorithm* located in PU_{Light} will be responsible for switching on/off the lights in each car using a dimmer.

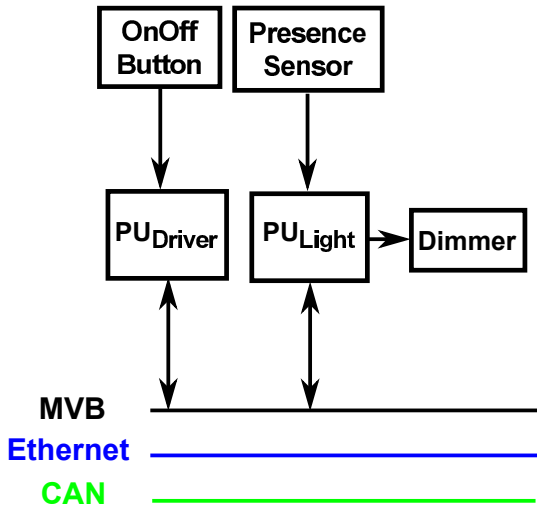


Figure 2.13: Hardware Model of the Light Control Function

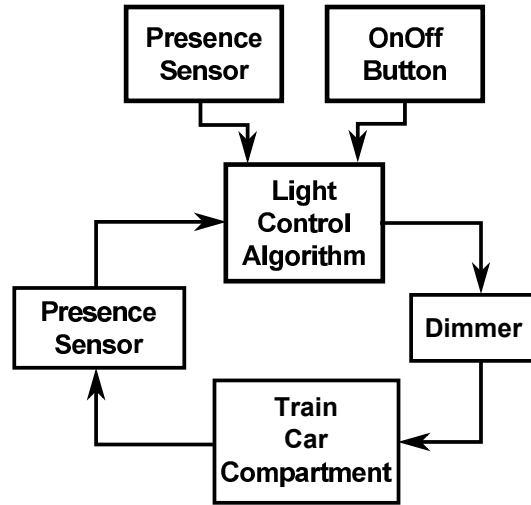


Figure 2.14: SW/Dependency Model of the Light Control Function

2.2 Dependability Framework

In this section we introduce the basic definitions for the development of this thesis, discuss about the essential fault tolerant design techniques and set the failure model of our approach, i.e., the failures that our approach is intended to deal with.

2.2.1 Dependability: Definitions and Classifications

The first fundamental definition concerns to dependability:

Definition 2.1. Dependability: *the ability to avoid failures that are more severe and more frequent than is acceptable [Avizienis04].*

What is acceptable for a system design is defined by the dependability requirements, which will limit its tolerable failures. Consequently, failure-related studies (also known as science of failures) will guide dependability determination and evaluation.

Every system is exposed to *threats*, while different *dependability mechanisms* are used to meet requirements. Dependability requirements are defined in terms of *dependability attributes*. This characteristics will be introduced in the following sections.

When designing a dependable system, the classification of fault, error and failure concepts, i.e., *dependability threats*, are fundamental so as to specify dependability requirements accurately.

Definition 2.2. *Fault*: *adjudged or hypothesized cause of an error. A fault is active if it produces an error, otherwise it remains dormant.*

Fault classification: elementary fault classes are grouped according to different viewpoints [[Avizienis04](#)]:

- *Phase of creation or occurrence of faults:* development faults emerge during the system development and operational faults appear during the system operation.
- *System boundaries:* internal faults and external faults resulting from the interaction with the physical or human environment.
- *Phenomenological causes:* natural/hardware faults and human-made faults.
- *Dimension:* hardware and software faults.
- *Objective:* malicious faults and non-malicious faults.
- *Developer's intent:* deliberate faults (bad decisions) and non-deliberate faults (mistakes).
- *Capability:* accidental faults and incompetence faults.
- *Persistence of faults:* permanent faults and transient faults.

Definition 2.3. *Error*: *part of the system's total state that may cause its subsequent failure. Errors are the responsible for deviation between the computed value and the correct value [[Rausand03](#)].*

Error classification: a formal definition and classification of errors is given by [[Powell95](#)] which characterizes system services by the tuple $\langle vs_i, ts_i \rangle$ where vs_i is the value of the service and ts_i is the time or instant of observation of the service s_i .

The *correctness* of the system service s_i is specified by correct content ($vs_i \in SV_i$) and time instant ($ts_i \in ST_i$) values where SV_i and ST_i are respectively the specified sets of values and times for the service item s_i . Accordingly, different errors are defined:

- *Arbitrary value error*: $vs_i \notin SV_i$
- *Arbitrary timing error*: $ts_i \notin ST_i$
- *Early timing error*: $ts_i < \min(ST_i)$
- *Late timing error*: $ts_i > \max(ST_i)$
- *Omission (infinitely late) error*: $ts_i = \infty$
- *Impromptu error*: $(vs_i \notin SV_i) \wedge (ts_i \notin ST_i)$

Definition 2.4. Failure: an event that occurs and provokes the transition of the correct service to incorrect service. Different forms of transitions are defined through failure modes.

Failure classification: a service is characterized by the value and time the service is delivered. The different ways that deviations occur are *failure modes* and each failure mode is categorized by the *failure severity*. Failure modes are characterized according to the following viewpoints:

- *Failure domain*: failures are classified according to *value* and *timing* failures [[Bondavalli90](#)]:
 - *Value* failures: incorrect value failures are further refined into *coarse* incorrect (detectable value failures), *subtle* incorrect (undetectable value failures) and *omission* value failures (no output when required).
 - *Timing* failures: incorrect timing failures are classified as *early*, *late* and *infinitely late* (omission) failures.

When both value and timing failures occur, failures are classified as:

- *Halt* failures: the service is halted.
- *Erratic* failures: the service is delivered but is erratic.
- *Detectability of failures*: *signalled* failures and *un-signalled* failures.
- *Consistency of failures*: *consistent* failures and *inconsistent* (random) failures.
- *Consequence/Criticality of failures*: the consequence of the failures are quantified

by the failure severities to which maximum acceptable probabilities of occurrence are associated. A common classification of failure severities include *catastrophic*, *critical*, *major* and *minor* consequences.

Assuming that a system is constituted by a set of interacting components, the state of the system will be determined by the state of its constituent components. The occurrence of a fault or combination of faults on hardware and software components provoke errors and when errors lead the system function to perform incorrectly, system failure occurs. Table 2.1 and Table 2.2 display the classification of faults and error/failures respectively.

Table 2.1: Fault Classification

Phenomenological Cause	Physical
	Human Made
System Boundaries	Internal
	External
Phase of Creation	Development
	Operational
Dimension	Hardware
	Software
Objective	Malicious
	Non-Malicious
Capability	Accidental
	Incompetence
Persistence	Permanent
	Transient

Table 2.2: Failure/Error Classification

Domain	Value	Coarse
		Subtle
		Omission
	Time	Early
		Late
		Omission
Consistency	Non-Consistent	
	Consistent	
Persistence	Transient	
	Permanent	
Detectability	Signalled	
	Un-signalled	

So as to specify dependability requirements, let us define *dependability attributes*:

Definition 2.5. Reliability: *ability of an item to perform a required function, under given environmental and operational conditions for a stated period of time [Rausand03]. Statistically: assuming \mathcal{X} represents the random variable which determines the time to failure of the system, reliability ($\mathcal{R}(t)$) is defined as the probability that the system will be successfully operating from time 0 to time t:*

$$\mathcal{R}(t) = \mathcal{P}(\mathcal{X} > t) \quad (2.1)$$

The failure probability or unreliability is then:

$$\mathcal{F}(t) = 1 - \mathcal{R}(t) = \mathcal{P}(\mathcal{X} \leq t) \quad (2.2)$$

Assuming non-repairable components, reliability is expressed informally as the probability of the system remains operative throughout a time interval. If the assumption of non-repairable components does not hold, reliability with repairs needs to be considered: probability of the system experiences no failures throughout a time interval given that it was operative at the initial time instant.

Definition 2.6. Mean Time To Failure: is defined as the expected value of the lifetime before a failure occurs. Statistically: assuming \mathcal{X} is the random variable that represents the time to failure and $f(t)$ the probability density function of the system lifetime ($f(t) = \frac{d\mathcal{F}(t)}{dt}$), the *MTTF* is defined as:

$$MTTF = E[\mathcal{X}] = \int_0^{\infty} tf(t)dt = \int_0^{\infty} \mathcal{R}(t)dt \quad (2.3)$$

Definition 2.7. Failure Rate Function (Hazard Function): the conditional probability of a component of age t failing in $(t, t + \Delta t]$ given that it has not failed in $[0, t]$. It indicates the changing rate in the ageing behaviour over the life of a population of components.

The probability that an item will fail in the time interval $(t, t+\Delta t]$ when we know that the item is functioning at time t is:

$$Pr(t < \mathcal{X} \leq t + \Delta t \mid \mathcal{X} > t) = \frac{Pr(t < \mathcal{X} \leq t + \Delta t)}{Pr(\mathcal{X} > t)} = \frac{\mathcal{F}(t + \Delta t) - \mathcal{F}(t)}{\mathcal{R}(t)} \quad (2.4)$$

By dividing Equation 2.4 by the length of the time interval, Δt and letting $\Delta t \rightarrow 0$, we get the failure rate function ($\lambda(t)$) of the item:

$$\lambda(t) = \lim_{\Delta t \rightarrow 0} \frac{Pr(t < \mathcal{X} \leq t + \Delta t \mid \mathcal{X} > t)}{\Delta t} = \lim_{\Delta t \rightarrow 0} \frac{\mathcal{F}(t + \Delta t) - \mathcal{F}(t)}{\Delta t} \frac{1}{\mathcal{R}(t)} = \frac{f(t)}{\mathcal{R}(t)} \quad (2.5)$$

Definition 2.8. Maintainability: ability to undergo repairs and modifications to restore or retain to a state in which can perform its required functions.

Informally, maintainability is the probability of isolating and repairing a fault in a system within a given time.

Definition 2.9. Mean Time To Repair: *is the expected value of the repair time. Statistically: let \mathcal{Y} to be the random variable that represents the time to repair of a system and $g(t)$ the density function of the system repair time, we define MTTR as:*

$$MTTR = E[\mathcal{Y}] = \int_0^{\infty} tg(t)dt \quad (2.6)$$

Availability comprehends both reliability and maintainability concepts:

Definition 2.10. Availability: *Operate correctly at a certain point in time when a service is requested [Rausand03]. Statistically: assuming $\mathcal{I}(t)$ is a Bernoulli random variable (1: operative; 0: failed) the point availability ($\mathcal{A}(t)$) is defined as:*

$$\mathcal{A}(t) = \mathcal{P}\{\mathcal{I}(t) = 1\} \quad (2.7)$$

Definition 2.11. Average Availability (A_{av}): *is defined in $[0, t]$ as*

$$\mathcal{A}_{av}(t) = \frac{\int_0^t \mathcal{A}(t)dt}{t} = \frac{MTTF}{MTTR + MTTF} \quad (2.8)$$

Definition 2.12. Safety: *absence of catastrophic consequences on the user(s) and the environment. The aim of safety analysis techniques is to evaluate whether a system meets its safety requirements. Safety requirements are defined as a hazard³ (i.e., injury or incidents) combined with the tolerable probability of this hazard [Leveson95].*

Since security aspects are outside of the scope of this thesis, we will not consider confidentiality and integrity as dependability attributes. Henceforth, throughout this dissertation the term dependability will focus on Reliability, Availability, Maintainability, and Safety (RAMS) attributes.

³Hazard is an state of the system, which may develop into an accident either through the factors that are not under the control of the system, uncontrollable external actions or through a sequence of normal events. It is the last decision point before an accident.

2.2.2 Designing for Fault Tolerance and Dependability

Based on the knowledge that faults are present within the system components, development of an appropriate system within specified constraints will be guided by the *dependability mechanisms* [Kaaniche02; Avizienis04]:

- *Fault Prevention*: prevent the occurrence or introduction of faults, e.g., project planning and risk assessment activities enable system's fault prevention.
- *Fault Removal*: reduction of the number and severity of faults including verification, diagnosis and modification activities.
- *Fault Forecasting*: estimation of the present number, future incidence and the likelihood of the consequence of faults. Fault forecasting activities include statement of the dependability objectives, allocation of the objectives and qualitative and quantitative evaluation to assess whether the system satisfies the objectives.
- *Fault Tolerance*: delivery of correct service in the presence of faults, e.g., selection of the adequate fault and error handling mechanisms.

Fault prevention and fault removal techniques are aimed at reducing system faults and both techniques are included in the *fault avoidance* paradigm. Fault forecasting and fault tolerance are embodied in the concept of *fault acceptance* based on the assumption that the design of a system without faults is not achievable. In order to design a dependable system, each of the four mechanisms need to be considered, but not necessarily as separate concepts. *Fault tolerance* concept encompasses all means by structuring the system so as to avoid faults. When inevitable faults occur, countermeasures are adopted in the form of redundancies to deliver correct service in the presence of faults.

Different architectural decisions influence both dependability and cost (e.g., see [Somani97; Nord03; Cortellessa06; Gokhale07]). Hardware costs, power requirements, processing time and weight of the added hardware elements (critical parameter for some fields such as avionics) are some design consequences that need to be considered when designing a system for fault tolerance. This situation leads to different design strategies such as optimal architecture selection based on the trade-off between dependability attributes, cost and complexity; or refinement of the structure until achieving an adequate compromise solution.

Fault tolerance mechanisms may overcome all dependability goals, but specially they are aimed at reducing the frequency of failures and mitigating their effects (failure avoidance). Designing a fault-tolerance strategy involves the following steps [Nelson90]:

1. *Error Detection*: takes place either during normal service delivery or while normal service delivery is suspended.
2. *Error Containment*: prevention of the propagation of erroneous information across defined boundaries.
3. *Error Masking*: dynamic correction of the error allowing the continuity of correct service in the presence of errors.
4. *Error Recovery*: systematic or on-demand correction of an erroneous system state. On-demand correction brings the system to a error-free state by applying techniques like backward or forward recovery. Error compensation uses redundancy within the erroneous state to mask errors on-demand or systematically.
5. *Fault Diagnosis*: identification of the module responsible for a detected error.
6. *Fault Repair/Reconfiguration*: exclude or replace the faulty component.
7. *Verification* of the effectiveness (or coverage) of the fault tolerant strategy.

Redundancy Classification

The key ingredient in fault tolerant techniques is redundancy, that is, the addition of information, time, or resources beyond what is needed for normal system operation. Different classes of redundancies are employed to achieve dependability requirements [Johnson84].

Informational redundancy is focused on providing additional information to the basic data structure. This redundancy can be used for: *error detection* with the aim to distinguish valid and invalid code words (e.g., error-detecting codes, checksums); *error correction* allowing the real-time computation without interruptions (e.g., Hamming, Reed Solomon error-correcting codes); or *error recovery* providing a fail-over recovery point through the implementation of a requirement function using diverse techniques

(analytical redundancies), e.g., calculation of the acceleration using different physical variables (e.g., position, speed) linked with their analytical relationships.

Hardware redundancy (also known as spatial redundancy) deals with the redundancy of physical resources and it can be classified in three forms of replication:

1. *Static or passive replication* masks predefined occurrence of faults to prevent their propagation using the concept of majority voting to determine the output of the system and do not offer detection, isolation or repair of a faulty module (e.g., Triple Modular Redundancy (TMR), see Figure 2.15).

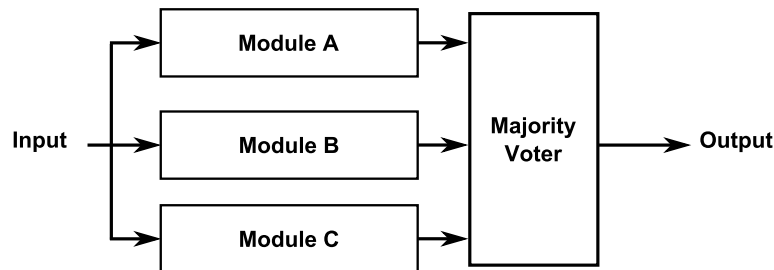


Figure 2.15: Triple Modular Redundancy Example

2. *Dynamic or active replication* does not mask faults, but detects and reconfigures faults so that a spare component can be switched to replace the faulty component (e.g., hot and cold spares).
3. *Hybrid replication* uses fault masking to prevent the fault from affecting the system and fault detection and reconfiguration to allow a spare module to replace the faulty component (e.g., N modular redundancy (NMR) with spares).

Temporal redundancy is based on redundant computations primarily used to distinguish between permanent and transient failures. To this end, multiple computations are performed with the aim to observe the behaviour of an error condition. Temporal redundancy aims to reduce the amount of extra hardware at the expenses of additional time (e.g., see [Agrawal88; Thuel94]).

Software redundancy adds extra software to provide the system with fault tolerance capabilities. Single-version and multiple-version software fault tolerance techniques are distinguished [Wilfredo00]. The former uses single version of a piece of software to detect and recover from faults and includes considerations on the software structure,

error detection and exception handling. The majority of single-version software recovery mechanisms implement checkpoint and restart strategies (either dynamic or static). The latter is characterized by the idea of building software components in different ways from a common specification (also known as design diversity - see [Littlewood01b] and references herein), in order to eliminate any sources of similar design faults. Examples of multi-version software fault tolerant techniques are recovery block, N version programming, N self-checking programming, and consensus recovery blocks techniques (see [Pullum01] for details of alternative strategies).

How redundancy is used in order to improve the dependability of the overall system is as important as the redundancy itself. An increase in the number of redundant elements does not guarantee better fault tolerance, instead it increases the overall failure probability. Effectiveness of fault tolerant architecture depends on the probability of common failures between its redundant parts [Littlewood01a]. To this end, diversity techniques are used by implementing alternative development/design techniques to create different redundant elements which may fail differently and protect the system against common cause failures (cf. Figure 2.16). While identical redundancies address random failures, diverse redundancies address both random and common cause failures.

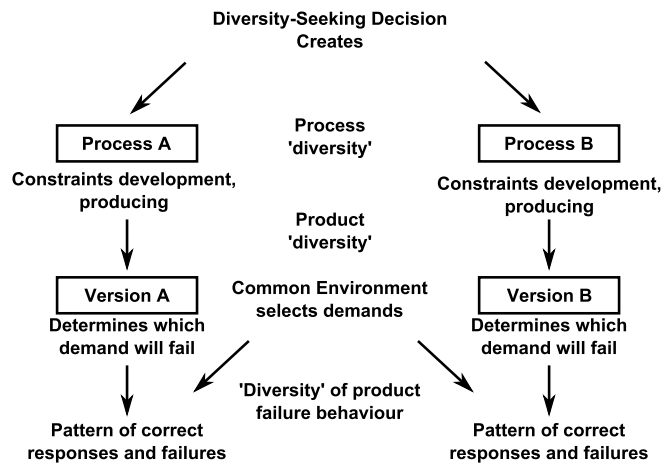


Figure 2.16: Diverse Design [Littlewood00a]

Theoretical models have been developed to evaluate the influence of diversity-seeking decisions and common cause failures on system reliability (e.g., see EL and LM models [Littlewood96]). Although these models are outside the scope of this work, it is worth mentioning that the quantitative evaluation of the influence of diversity on dependability is not a trivial task. In the well known example developed by Knight and Leveson

[Knight86], they tested empirically the assumption of statistical independence in N version programming. Their results show that there is correlation between independently developed versions and therefore, the assumption of independent errors does not hold.

We focus on a subset of design diversity techniques: *functional diversity* [Burlando92]. Functional diversity is a methodology consisting of N different implementations of the same requirement specification where each implementation uses a different input set and different algorithms to compute the same required output. With respect to normal diversity (e.g., N version programming), the basic differences are the followings: in functional diversity, N teams begin to work separately having as only common point the system requirements. The approaches to the problem and input data are different. In normal diversity, the N teams begin to work separately only after the specification has been written. Then each team use a similar approach: the same modelling of the process and the same data types; the differences among them lie only in the implementation techniques and in the details of the algorithms. In everyday systems there exist many applications which make use of functional diversity and implement *diverse redundancy* applications (i.e., deploying diverse replicas): cars have duplicated braking systems comprised of foot brakes and handbrakes or laptops have diverse backups for electrical power supply such as batteries.

The basic requirement to apply functional diversity is that the problem should be approached from different viewpoints, which leads to defining the major drawback of functional diversity: the need of an (brainstorming-like) intellectual process to obtain diverse specifications with the cost that this process incurs. Therefore, an important issue which needs to be addressed when undertaking functional diversity is whether the fault tolerance will produce enough reliability (dependability) gain to be worth its cost.

2.2.3 Fault Hypothesis & Failure/Error Model

Fault assumptions are closely related to the fault-tolerance management decisions. Any assumption which does not adhere to the real operation of the system will cause an overall decrease on the system dependability. Therefore, it is necessary to define which faults the system is going to tolerate, i.e., *fault hypothesis*, and arrange them in a *failure/error model* so as to characterize the possible fault recovery strategies systematically.

Table 2.3 describes the fault hypothesis of the systems that we will deal with and Table 2.4 displays the failure/error model of the system. Examples of the faults that we plan to address with this dissertation are: permanent software development defects or hardware deteriorations; development faults and faults which emerge from designers incompetences and accidents. The influence of accidental and incompetence faults will be considered by assigning a failure rate to the human-made software resources. Thus, the considered human-made faults will cover software development faults, but we will not address the influence of human faults as is. External environmental influences will not be addressed neither, we will consider only system’s internal faults such as hardware, software or communication resources faults. Since we are not dealing with security issues, malicious faults will not be contemplated as well.

Table 2.3: Fault Hypothesis

Phenomenological Cause	Physical	✓
	Human Made	X
System Boundaries	Internal	✓
	External	X
Phase of Creation	Development	✓
	Operational	X
Dimension	Hardware	✓
	Software	✓
Objective	Malicious	X
	Non-Malicious	✓
Capability	Accidental	✓
	Incompetence	✓
Persistence	Permanent	✓
	Transient	X

Table 2.4: Failure/Error Model

Domain	Value	Coarse	✓
		Subtle	✓
		Omission	✓
	Time	Early	✓
		Late	✓
		Omission	✓
Consistency	Non-Consistent	X	
	Consistent	✓	
Persistence	Transient	X	
	Permanent	✓	
Detectability	Signalled	✓	
	Un-signalled	X	

Fault detection techniques are necessary to detect the presence of these faults. According to the failure/error model (cf. Table 2.4), for simplicity we will assume consistent and permanent failures. The addressed fault detection strategies are based on time and value thresholds either as statically predetermined or dynamically determined thresholds (model-based fault detection, e.g., see [Isermann05] and references herein).

Redundancy Model

Design strategies (redundancies) differ when managing different kind of failures. From the different classes of redundancies, we focus on hardware, software, and information redundancies implemented as follows:

- Hardware redundancy with dynamic redundancy strategies provides fail-over capabilities by detecting faults and reconfiguring the system behaviour and/or system structure to adapt the system operation.
- Software redundancy focuses on the strategical distribution of system functionality among different PUs to repair the system functionality in the presence of failures.
- Information redundancy is used to provide compatible functionalities by reusing and/or adapting already existing information in a system.

In order to simplify the nomenclature when dealing with hardware, software, and information redundancies, based on the possibility of reusing hardware resources through compatible functionalities, we define two kind of redundancies:

Definition 2.13. *Homogeneous redundancies*: redundancies which explicitly replicate the nominal functionality making use of additional explicit hardware components (and hence, software modules and information sources), e.g., N modular redundancy.

Definition 2.14. *Heterogeneous redundancies* [Shelton04; Wysocki07]: redundancies which reuse existing hardware resources in a system and provide a compatible functionality (i.e., emerge from heterogeneous functionalities) with the information that already exists in the system, e.g., analytical redundancy.

2.2.4 Opportunity Analysis

Our design goal focuses on designing for redundancy instead of designing for failure diversity [Strigini05]. Designing for failure diversity focuses on adding diversity-related design approaches deliberately in order to improve fault tolerance. Our work focuses on adding redundancies (and required fault tolerant mechanisms) where deemed necessary by exploiting implicit diversity which may exist in some specific environments

(see Section 1.2) in order to provide fault tolerance and reduce costs. We do not focus directly on evaluating the influence of diversity-seeking decisions on dependability, but eventually this may happen as a side-effect of our design goals.

The concept of functional diversity is generalist and directly aligned with analytic redundancies in that both approaches use diverse algorithms to produce equivalent results. Despite not following directly a diversity-seeking decision approach our method make use of diverse functions and fault-tolerant choices that affect these systems: number of redundancies, fault detection and reconfiguration methods, and allocation of software modules to the hardware modules. The primary concern of the designers when adding fault-tolerant strategies should be to manage the complexity resulted from the need to manage additional resources and corresponding mechanisms. Therefore, it is necessary to adopt trade-off decisions between the incurred cost and attained fault tolerance (and dependability) level.

In the scientific literature there have been two different viewpoints towards the concept of heterogeneous (and homogeneous) redundancies: (1) redundancy allocation problems (e.g., see [yangLi10; Sharma11]) have considered as heterogeneous redundancies those components which have different characteristics (e.g., memory, processing power). (2) In [Shelton04], Shelton and Koopman introduced the heterogeneous redundancy concept as an approach to reduce costs through analytic redundancy like techniques.

Aligned with the idea of Shelton and Koopman, our concept of heterogeneous redundancies encompass a general class of redundancies. These are: analytical redundancies, redundancies which emerge from some functional diversity applications and redundancies arising from overlapped system functions in massively networked scenarios [Aizpurua13a]. Since the reuse of hardware elements can emerge in different situations heterogeneous redundancies can take many forms:

- Alternative algorithms providing an equivalent output, e.g., analytic redundancies: alternative equations by linking sensors/actuators in different ways so that they can provide additional (heterogeneous) functions.
- Implementations/functions located in alternative contexts able to provide compatible implementations to other implementations/functions, e.g., temperature sensors located in contiguous compartments able to provide compatible functionalities to each other compartments (reuse of the temperature sensor).

- Alternative functionalities able to provide compatible implementations to other functions, e.g., Video Surveillance function may provide a compatible function to the Door Status Control function by adding a image processing SW to the camera (reuse of the camera).

Our goal with the use of heterogeneous redundancies is not only to reduce hardware costs, but also to maintain or even improve the dependability level of the system design. Heterogeneous redundancies include overlapping system functions which may add diversity to the system architecture due to the inherent properties of a networked control system operating in massively networked scenarios. The intuition of the advantages provided by the use of heterogeneous redundancies need to be demonstrated quantitatively so that it is possible to adopt trade-off decisions between dependability and cost when deciding to implement alternative redundancy techniques.

Motivated by these issues we outline a design approach which evaluates systematically the influence of fault tolerance and diversity-related design decisions on system dependability and cost (see Chapter 3). To do so, a dependability evaluation algorithm and further analyses such as the sensitivity evaluation of redundancies have been implemented (see Chapter 4 and Chapter 5).

2.3 Overview of the Main Dependability Analysis Approaches

Dependability analysis techniques can be organised by looking at how different system failures are characterized with its corresponding underlying formalisms. On one hand, *event-based* approaches reflect the system failure behaviour and structural relationships through combination of events. This analysis results in either Fault Tree (FT) like [Vesely02] or Reliability Block Diagram (RBD) like [Rausand03] structures, which emphasizes the reliability and safety attributes. On the other hand, *state-based* approaches map the analysis models into state-based formalisms such as Markov chain or Petri nets. These approaches analyse system changes with respect to time and mainly concentrate on reliability and availability attributes. Interested readers refer to the Appendix A for basic definitions of event-based and state-based formalisms.

This section has been divided into 2 subsection: Subsection 2.3.1 overviews extensions of event-based approaches and combinations of both event-based and state-based approaches - *hybrid approaches*; and Subsection 2.3.2 evaluates the utility of *hybrid approaches* to evaluate complex systems - opportunity analysis.

2.3.1 Hybrid Approaches

Hybrid approaches overcome the main limitations of event-based approaches and provide mechanisms to address some of the drawbacks arising from state-based approaches.

The extended usage of event-based approaches for dependability-related tasks have lead to the identification of their main limitations, see Table 2.5.

Table 2.5: Limitations of Event-Based Approaches [Aizpurua13b]

ID	Limitation
<i>L1</i>	Event-based approaches are static representations of the system, neither time information nor sequence dependencies are taken into account [Dugan92].
<i>L2</i>	The orientation of the event-based approaches concentrates on the analysis of failure chain information. Consequently, their hierarchy reflects failure influences without considering system functional architecture (design) information [Kaiser03].
<i>L3</i>	Event-based (and state-based) quality evaluation models depend on the analyst's skill to reflect the aspects of interest. Failure modes and undesired events must be foreseen, resulting in a process highly dependent on analyst's knowledge of the system [Galloway02].
<i>L4</i>	Manageability and legibility of event-based (and state-based) quality evaluation models is hampered when analysing complex systems. Model size, lack of resources to handle interrelated failures and repeated events/components, in conjunction with few reusability means, are its main impediments [Kaiser03] [Price02].

L1 refers to the capability of the technique to handle temporal notions. This is of paramount importance when analysing fault tolerant systems taking into account system *dynamics* such as load sharing, standby redundancy, on-demand failures, dependent failures, cascade failures or common cause failures.

L2 emphasizes the interdisciplinary work between dependability analysis and architectural design. Joining both procedures helps obtaining a design, which meets depend-

ability requirements consistently.

L3 entails a trade-off solution between the time consuming analysis resulting from understanding the failure behaviour of the system and the acquired experience. A substantial body of works have focused on the automatic generation of analysis models from design models addressing limitations L2 and L3 (refer to groups 3, 5 in Appendix B, Table B.1). These approaches reuse design models showing the effects of design changes in the analysis results. However, the correctness of the analysis lies in the accuracy of the failure annotations.

Finally, L4 underlines the capability of the dependability analysis model to handle the component-wise nature of embedded systems. This permits obtaining a model that better adheres to the real problem and avoids confusing results.

Many authors have developed new alternatives or extended existing ones. Three groups are identified in order to gather the *hybrid approaches* with respect to the limitations they address:

- L1 is addressed in the Subsection *Dynamic Solutions for Static-Logic Approaches*.
- L2 and L4 are covered in the Subsection *Compositional Failure Propagation Analysis Approaches*.
- Specifically focusing on L3 and generally addressing the remainder of limitations *Model-Based Transformational Approaches* are studied.

Note that some approaches cannot be limited to a specific group, hence they are classified accordingly to its main contribution.

Dynamic Solutions for Static-Logic Approaches

The limitation concerning the lack of temporal and dependency information has been addressed by several authors to deal with system *dynamics* such as redundancy or repair strategies. Specific solutions for event-based FT and RBD approaches and solutions which combine event-based and state-based approaches have been proposed.

Fault Tree extensions: [Dugan92] introduced the Dynamic Fault Tree (DFT) methodology to address the analysis of configuration changes. DFTs were conceived

to model the reliability of systems which pose complex dependencies. New gates were added (dynamic gates) to the traditional (static) Fault Tree definition (see Figure 2.17):

- **Priority AND (PAND)** gate: $Y = PAND(E_1, E_2, \dots, E_N)$; Y is *true* iff all events $\{E_1, E_2, \dots, E_N\}$ are true and occur in the following order: $E_1 \triangleleft E_2 \triangleleft \dots \triangleleft E_N$; otherwise is *false* (cf. Figure 2.17 (a)).
- **Functional Dependency (FDEP)** gate: $[E_1, E_2, \dots, E_N] = FDEP(T)$; $\{E_1, E_2, \dots, E_N\}$ are *true* if the trigger event T occurs or they fail by themselves; otherwise they are *false* (cf. Figure 2.17 (b)).
- **Sequence Enforcing (SEQ)** gate: $SEQ(E_1, E_2, \dots, E_N)$; $\{E_1, E_2, \dots, E_N\}$ are true iff all events $\{E_1, E_2, \dots, E_N\}$ are true and occur in the following order: $E_1 \triangleleft E_2 \triangleleft \dots \triangleleft E_N$; otherwise they are *false* (cf. Figure 2.17 (c)). Input events are forced to fail in a particular order and different failure sequences can never take place.
- **Spare (SP)** gate: $Y = SP(E_{Act1}, E_{sp1}, E_{sp2}, \dots, E_{spN})$; Y is *true* iff the active event E_{Act1} and all spares $\{E_{sp1}, E_{sp2}, \dots, E_{spN}\}$ have failed, otherwise is *false*. Spares may be in any of the following states: stand-by, working or failed. Spares can fail in working and stand-by states: λ_{Act_j} is the failure rate of the spare that is in working state, $\alpha_{Act_j} \lambda_{Act_j}$ is its failure rate in the dormant state (cf. Figure 2.17 (d)).

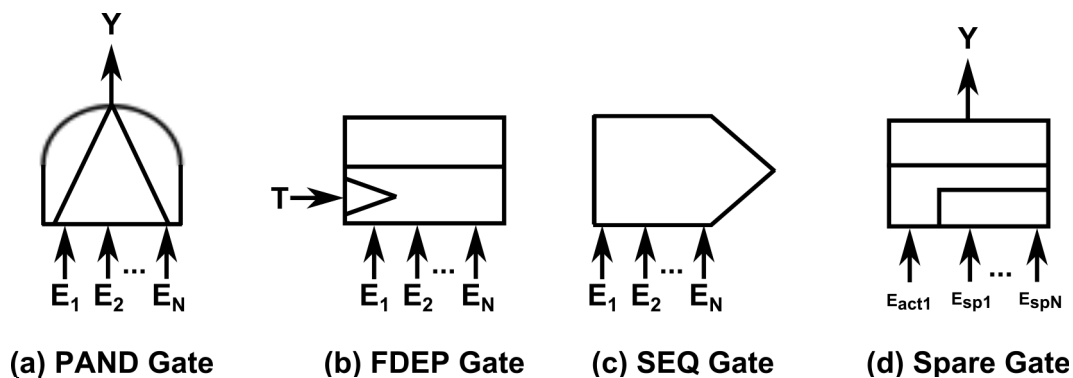


Figure 2.17: Dynamic Fault Tree Symbols

DFT models can be solved *analytically* or via *simulation* [Chiacchio11]. To solve DFT models *analytically* it is necessary to transform the DFT model into its equivalent stochastic model. If basic events are characterized by the exponential distribution, the DFT model can be mapped to a Continuous Time Markov Chain (CTMC) directly and solve it analytically by using differential equations. Other alternatives to solve a

DFT model address:

- *Generalized Stochastic Petri Nets (GSPN)* [Bobbio04] [Codetta-Raiteri05]: Dynamic Repairable Parametric Fault Tree (DRPFT) approach allows a compact representation of the system including repairable basic events. It enables folding identical sub-trees in a single parametric sub-tree (cf. Figure 2.18). System's unreliability is obtained by solving the Stochastic Well Formed Net model [Bobbio04] or through the evaluation of GSPN models [Codetta-Raiteri05]. The approach relies on exponential failure/repair distributions.
- *Dynamic Bayesian networks* [Montani08; Portinale10]: avoids the state-explosion problem by transforming the DFT model into a Dynamic Bayesian Networks (DBN) model. DBN is a stochastic transition model factored over a number of random variables. Discrete time is used to cope with the high computational effort arising from exact time-continuous calculations. It supports the analysis of repairable events and components through repair box gates [Portinale10].
- *Input/Output interactive Markov chains* [Arnold13]: provides a compact representation of the system and it supports exponential and phase-type distributions. The I/O interactive Markov chain is a compositional CTMC and it reduces the final state-space. It is analysed through stochastic model checking and repairable basic events are not addressed (cf. Figure 2.19).
- *Structure function* [Merle14]: presents an algebraic framework to extract the structure function of a DFT and calculate the exact solution of systems independent of the failure distribution. The approach requires high computational effort even for small systems, it is not yet implemented, and it is unable to deal with repairable basic events (see Appendix C.5 for an example).

For complex systems the traceability from the DFT model to the state-based analysis model and vice-versa is difficult to follow due to the flat characteristics of the DFT model and its state-based analysis model. [Bobbio04; Codetta-Raiteri05] presented the DRPFT approach to deal with the manageability issue of representing several replicas in a Fault Tree by taking advantage of symmetric DFT configurations. Sub-trees linked with the same gates and same failure rates are folded and parametrized (cf. Figure 2.18). However, its underlying analysis model (GSPN) is a flat state-space model. Furthermore,

if the same subsystem does not fail with the same logic it is not parametrizable (e.g., see Figure C.4).

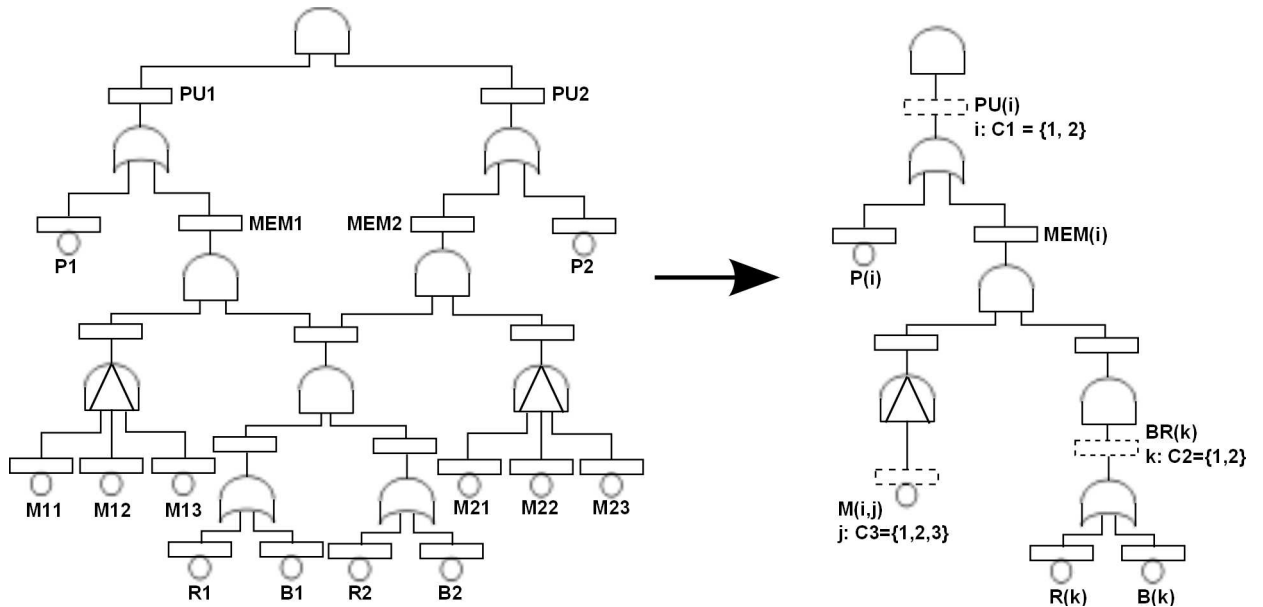


Figure 2.18: Dynamic Parametric Fault Tree Example

[Arnold13] overcome the flatness of the dependability analysis model through compositional Markov chain. Although the analysis model is compositional, the DFT model suffers from flatness. When the size of state-space model or DFT model increases, it becomes error prone and difficult to maintain (see Appendix C).

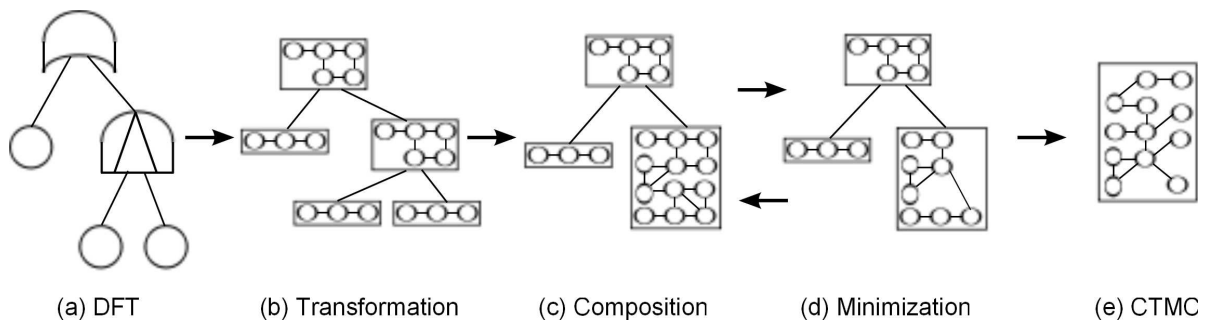


Figure 2.19: Composition Aggregation Method of [Arnold13]

Other alternatives to analyse DFT models are based on *simulations*:

- [Rao09] and [Manno12b] implement Monte Carlo simulations to address any distribution of basic events. To this end, multiple computations are performed over

the DFT model characterized with the failure logic of the DFT gates and random variables representing the failure times of basic events.

- [Manno14c] introduced a discrete event simulation approach based on Adaptive Transition System paradigm [Manno12a]. This promising approach is able to capture any distribution as well as repair characterizations of basic events.

Simulation approaches have been successfully applied to address size, repair behaviour and statistical distribution assumptions/limitations. Their drawback comes from the required calculation time, which increases with the required accuracy of the results. See Chapter 4, Subsection 4.2.4 for further details and implementation of Dynamic Fault Trees using Monte Carlo simulations.

Reliability Block Diagram extensions: following the way of DFTs, an approach emerged based on *dynamic* RBDs.

Dynamic RBDs (DRBDs) [Distefano07; Distefano09] model failures and repairs of components through the specification of state machines for each component and inter-component cause/effect dependencies. To analyse DRBD models quantitatively, these are transformed into GSPN models and its underlying CTMC is obtained and solved using the WebSPN tool [Puliafito14]. Another solution to solve DRBD models was presented in [Robidoux10] through the conversion of DRBDs into Coloured Petri Nets. However, to the best of our knowledge, an integrated modelling and analysis toolset for DRBDs is lacking.

[Signoret13] presented an approach called Reliability Block Diagram driven Petri nets (RdP) which uses RBDs as an interface to build large Petri nets systematically. The modular characterization of Petri nets enables the intuitive creation of RdP models from predefined module libraries.

Aligned with these formalisms, the OpenSESAME modelling environment connects RBDs and state-based formalisms [Walter08]: its input models are based on RBDs and failure dependency diagrams, while component tables and repair tables are used to indicate component-specific failure/repair characteristics and inter-component dependencies. To perform the quantitative analysis of the system, OpenSESAME models are transformed into SPN and Stochastic Process Algebra models.

Combination of event-based and state-based approaches: progres-

sion in the conjoint use of event-based and state-based formalisms is reflected with Boolean logic Driven Markov Process (BDMP) [Bouissou07] and State-Event Fault Tree (SEFT) [Kaiser07] formalisms.

BDMP employs static FT as a structure function of the system and associates Markov processes (or Petri nets, if necessary) to each leaf of the tree. Triggers are used to influence the occurrences of basic events or gates with the failure occurrence of other basic events or gates existing in the same FT. BDMP enables modelling the repair behaviour of basic events as well as sequences of failure events.

However, the main limitations of BDMP come from the trigger event itself: (1) the failure of the trigger event is not taken into account and (2) the trigger is able to consider only two processes while in some cases it is necessary to use more processes to fully describe the behaviour of the system.

SEFT formalism combines FT elements with both Statecharts [Harel87] and Markov chains, increasing the expressiveness of the model. SEFT deals with functional and failure behaviour, accounts for repeated states and events, where the events are characterized as deterministic and/or exponentially distributed events, and allows the automatic transformation of SEFT models into Deterministic and Stochastic Petri Nets (DSPN) models for state-based analysis. Besides, the SEFT model allows modelling the system compositionally by linking components in a FT-like structure while managing system's complexity (refer to Appendix C for a SEFT model example and more details).

Dynamic Fault Tree is a well-known mature approach for the evaluation of the system's *dynamics*. It has been widely implemented over the last years (see Appendix B Table B.2 for the tool support) and different extensions have been performed: due to the properties of the CTMCs, the use of Dynamic Fault Trees has been limited to model events characterized with exponential distributions. This fact have awakened the scientific community to develop alternative analysis formalisms so that it is possible to model any failure distributions (e.g., [Rao09],[Manno12b], [Manno14c], [Merle14]). Moreover, DFTs were originally conceived to evaluate the unreliability of systems, but there have been many extensions in order to include repairable basic events and evaluate systems unavailability, e.g., repairable DFT [Manno14c], BDMP [Bouissou07], SEFT [Kaiser07], Radyban [Portinale10], DRBD [Distefano09].

As a result of the combination of state-based and event-based approaches to solve Dy-

dynamic Fault Tree models, specific problems of state-based approaches have emerged among Dynamic Fault Tree solutions which use state-based models for its resolution. The main impediments are the state-explosion problem and difficulty to understand the analysis model intuitively.

To improve the manageability, maintainability and traceability of these models, component-based characterizations have been suggested [Kaiser07]. Although limited in modelling capabilities, the compositional and transformational features of the SEFT approach, provide an adequate abstraction of the system structure and behaviour. Since the underlying analysis model of SEFTs is based on Deterministic and Stochastic Petri Nets, it may suffer from manageability issues (flatness) and it is limited to deterministic and exponentially distributed events.

Although the component-based characterization have been applied to SEFT models, due to the SEFT model's limitations there is room to extend the compositional paradigm towards the dynamic analysis of systems. Indeed, the component-based characterization has not been integrated with Dynamic Fault Tree models yet (see Chapter 4).

Compositional Failure Propagation Analysis Approaches

The main objective of Compositional Failure Propagation (CFP) approaches is to avoid unexpected consequences resulting from the failure generation, propagation, and transformation of components. Common factors for CFP approaches are:

- Characterization of the system architectures by design components.
- Annotation of the failure behaviour of each of component constituting the system.
- System failure analysis based on inter-components influences.

CFP approaches characterise the system as component-wise developed FT-like models linked with a causality chain. System architectural specifications and subsequent dependability analyses of CFP approaches rely on a hierarchical system model. This model comprises components composed from subcomponents specifying system structure and/or behaviour. CFP approaches analyse the system failure behaviour through characterizations of individual components, which lead to achieving a manageable failure analysis procedure.

Failure Propagation and Transformation Notation (FPTN) [Fenelon93], Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) [Papadopoulos11] and Component Fault Tree (CFT) [Kaiser03] are the principal CFP approaches. Their main difference is in the failure annotations of components, which specify incoming, outgoing and internal failures to each component: (1) FPTN uses logical equations, (2) HiP-HOPS makes annotations using Interface Focused Failure Mode and Effect Analysis (FMEA) (IF-FMEA) tables and (3) CFT associates to each component individual FTs. Subsequently, the connections between system components determines the *failure flow* of the system, linking related failure annotations of each component.

Concerning the different contributions of CFP approaches, FPTN first addressed the integration of system-level deductive FTA (from known effects to unknown causes) with component-level inductive FMEA (from known causes to unknown effects).

HiP-HOPS integrates design and dependability analysis concepts within a hierarchical system model. However, instead of exclusively linking functional components with their failure propagations like in FPTN, first the hierarchical system model is specified and then, compositional failure annotations are added to each component by means of IF-FMEA annotations. These annotations describe the failure propagation of component in terms of outgoing failures specified as logical combinations of incoming and internal failures (cf. Figure 2.20).

From the IF-FMEA annotations shown in Figure 2.20, the outgoing failures at the port *out_1* will be specified as follows:

$$omission-out_1 = omission-in_1 \text{ AND } omission-in_2 \text{ OR } Stuck \text{ at } 0$$

Once all the outgoing failures of all the ports are characterized, a FT synthesis algorithm analyses the propagation of failures between connected components. Traversing the hierarchical system model, while parsing systematically the IF-FMEA annotations of its constituent components, allows the extraction of the system FT and FMEA models.

CFTs are a model-based extension of FTA models. The component FTs can be combined and reused to systematically obtain the FT for any failure without having to create and annotate a FT for each failure. In order to integrate analysis and design concepts, it has

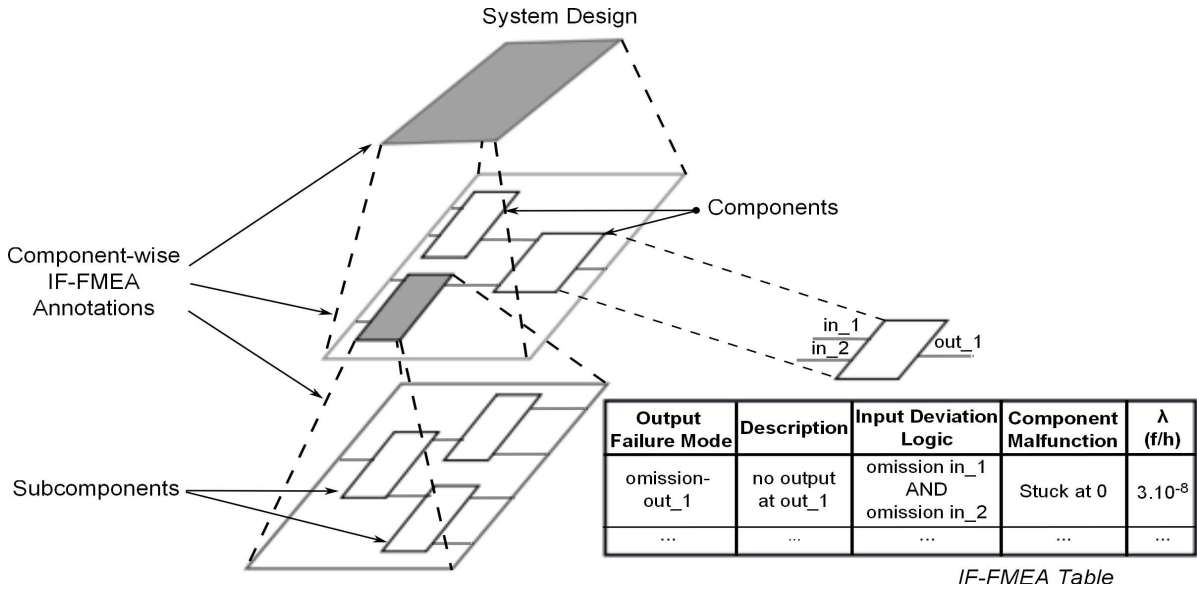


Figure 2.20: Hierarchical Structure and CFP Annotations in HiP-HOPS

been extended in [Domis09b] resulting in the Safe Component Model approach. The approach separates components' functional/failure specification and realization views and through the integration of the failure propagation and hierarchical abstraction, Safe Component Model allows obtaining a hierarchical component based abstraction of CFTs.

They all have been extended to cope with occurrences of **temporal events** influenced by the DFT approach. Temporal extensions for FPTN [Niu11] and HiP-HOPS [Walker09] concentrate on non-repairable systems by examining the order of events to identify sequence of events leading to the system failure, i.e., minimal cut-sequence sets.

Namely, the temporal extension of HiP-HOPS is based on the Pandora approach [Walker09]: it enables the dynamic qualitative analysis of event sequences through cut-sequence sets. Temporal Fault Tree (TFT) gates are defined to model complex time-dependent circumstances. For the quantitative analysis, algebraic models and Monte Carlo simulations are used with the TFT gates: priority OR - output occurs if: the first input occurs before the second input and the second input is not needed to occur [Edifor12]; simultaneous AND - all input events occur at the same time; and parametrized SAND - output occurs if two events happen within a given interval of time [Edifor13].

Within the approaches extended to deal with temporal events, there have been approaches which have been focused on **connecting CFP approaches with state-based approaches**: integration of CFT concepts with state-based techniques resulted in the SEFT formalism, which is able to handle availability and maintainability properties of repairable systems. Besides, HiP-HOPS has been connected with state machine specifications to generate (temporal) Fault Trees from state machine models [Mahmud12]. From individual state machines, Pandora failure expressions are generated transforming component-based state machines into TFT expressions of the system failures.

Other interesting extensions include mechanisms to **automate and reuse** analysis concepts: Failure Propagation and Transformation Calculus (FPTC) [Paige08a] approach adds the characterization of the nominal behaviour to FPTN models and generalizes the FPTN equations to improve the manageability and analysability. Moreover, an algorithm is implemented to cope with cyclic dependencies of feedback structures. In [Wolforth10], general failure logic annotation patterns were defined for HiP-HOPS. Similarly, the CFP approach presented by [Priesterjahn11a] emphasizes the reuse of failure propagation properties specified at the port level of components. These specifications focus on the physical properties of different types of flows, which allow reusing failure behaviour patterns for functional architectures.

The evolution of CFP approaches focus on reusability, automation and transformation properties. Since the annotations of the failure behaviour of components depend upon designers experience, reusing failure annotations leads to reducing the error proneness. Based on the knowledge that different dependability analyses have to be performed when designing a system, definition of a unique consistent model covering all analyses would benefit these approaches. This is why recent publications in this field centre on integrating existing approaches (see next Subsection). Interested readers please refer to Appendix B Table B.3 to see the tool support of the CFP approaches.

Combinations of dynamic dependability evaluation models able to model system dynamics and the compositional failure propagation approaches would result in a approach which is able to model repairable systems compositionally according to any failure distribution. Thus, motivated by this issue, a component-based approach for DFTs is defined in Chapter 4: Component Dynamic Fault Trees (CDFT).

Model-Based Transformational Approaches

Designing a dependable system presents many challenges throughout the development phase - from system specification to system validation and verification. This process is further complicated due to the increasing complexity of the current systems, which use many and different components. Model-based design approaches provide mechanisms to manage this complexity effectively.

Model-based transformational approaches were proposed to bridge the gap between design and analysis activities. Their main goal is to construct target dependability analysis models (semi-)automatically from source design models. The modelling process of transformational approaches is constituted of the following main activities:

1. The process starts from a compositional design description by using computer science modelling techniques.
2. The failure behaviour is specified either by extending explicitly the design model or developing a separate model, which is allocated to the design model - extended design model.
3. Transformation rules and algorithms extract dependability analysis models from the extended design model.

Architectural Description Languages (ADLs) provide an adequate abstraction to manage the system complexity [Medvidovic00]: Simulink [MathWorks14], Architecture Analysis and Design Language (AADL) [Feiler07] and Unified Modelling Language (UML) [OMG14b] have been used for both architectural and failure specification. UML is a widely used modelling language, which has been extended for dependability analyses following model-driven architecture concepts [OMG03]. Namely, profiles allow extending and customizing modelling mechanisms to the dependability domain [Fuentes04].

Lately, a wide variety of independently developed extensions and profiles have been proposed for dependability analysis [Bernardi12]. However, some generally applicable metamodel is lacking. In an effort to provide a consistent profile CHES ML emerged [Montecchi11]. CHES ML provides all necessary mechanisms to model dependable systems and extract either event-based (FMECA, FPTC) or state-based (SPN) models.

Many analysis approaches have been shifted towards the model-based transformational paradigm. Translations from high-level architectural description languages to well established compositional failure propagation analysis techniques, enable an early dependability analysis and allow undertaking timely design decisions, e.g.:

- The toolset for FPTC approach [Paige08a] relies on a generic metamodel in order to support transformations from SysML and AADL models.
- [Adler10a] developed a metamodel to extract CFT models from functional architecture models specified in UML. This process permits the generation of reusable CFT models consistent with the design model.
- In the same line, integration of HiP-HOPS model with EAST-ADL2 automotive UML profile is presented in [Biehl10].

AADL captures the system architectural model in terms of components and their interactions describing functional, mapping and timing properties. The core language can be extended to meet specific requirements with annex libraries. Behaviour and error model annexes are provided with the tool. The error annex links system architecture components to their failure behaviour specification making possible the analysis of the dependability attributes of the system. It has been used for event-based (FT) [Joshi07] and state-based (GSPN) [Rugina07] analysis.

AltaRica [Arnold99; Batteux13] is a dependability language, which enables describing the behaviour of systems when faults occur. The model is composed of several components linked together representing an automaton of all possible behaviour scenarios, including those cases when reconfigurations occur due to the occurrence of a failure [Romain07]. It is possible to process such models by other tools for model-checking, generation of FTs [Bieber02], Markov Chain generation, Petri Nets generation, or even for the generation of Boolean-Driven Markov Process models [Labri14].

[Riedl12] presented a language for the specification of reconfigurable and dependable systems called LARES. It expresses system's fault tolerant behaviour using a generic language in which any kind of discrete-event stochastic system can be specified. It is based on fully automated model transformations to measure systems dependability. Namely, transformations into TimeNET [TU Berlin07] and CASPA [Riedl08] tools are carried out in order to solve state-based stochastic Petri nets and stochastic process

algebra models respectively.

[Cressent11] defined a method for RAMS analysis centred on SysML [OMG14a] from where a FMEA model is deduced. SysML diagrams define a functional model connected to a dysfunctional database enabling the identification of failure modes. This database contains the link between system architecture and failure behaviour giving the key for FMEA extraction. Further, the methodology for dependability assessment is extended by using AltaRica, AADL and Simulink models. The approach addresses reliability analysis, timing analysis, and simulation of the effects of faults respectively.

Definition of a model for the extraction of all necessary formalisms for a complete/exhaustive dependability analysis is the common goal for the approaches included in this section. Interconnections between different formalisms in order to take advantage of the strengths of each ADL, allow analysing dependability properties accurately. AltaRica and AADL cover adequately the analysis of reliability, availability and maintainability attributes. Extraction of the main CFP approaches from ADLs should help to analyse comprehensively system safety properties. Moreover, Simulink model simulations allow evaluating the effects of failure and repair events in the system. Thereby, integrations between language specific models like in [Cressent11] helps evaluating accurately all dependability aspects of a system. The acceptance of the transformational approaches depends on the availability of tool-sets capable of performing (automatic) transformations. Interested readers refer to Appendix B Table B.4 to see the tool-support of the transformational approaches.

These approaches lead to adopting trade-off decisions between dependability design and analysis processes. On one hand, the automation and reuse of analysis techniques in a manageable way makes it a worthwhile approach for design purposes. The impact of design changes on dependability attributes are analysed systematically. On the other hand, from purist's point of view of classical analysis techniques, the automation process removes the ability of these techniques to identify and analyse hazards or malfunctions in a comprehensive and structured way.

Motivated by the lack of model-based solutions to identify heterogeneous redundancies systematically and evaluate their influence of system's dependability level automatically, in [Aizpurua13a] we presented a model-based solution to evaluate the failure probability of systems which use heterogeneous redundancies systematically.

Interested readers refer to Appendix B Table B.1 to see a classification of the analysed hybrid approaches based on the addressed limitations displayed in Table 2.5.

2.3.2 Opportunity Analysis

In order to classify (dynamic) Fault Tree - related approaches used in this section we will take into account: (1) their capability to model dynamic system configurations; (2) their possibility to characterize system's failure behaviour using the component-based paradigm; (3) their possibility to characterize the repair behaviour of the basic events of the system; and (4) their possibility to model any failure/repair distributions (cf. Table 2.6). Using a illustrative example, the model of each approach has been created in Appendix C in order to highlight their main characteristic.

Table 2.6: Addressed Characteristics by the Analysed Approaches

Approach	Dynamic	Component Based	Repair	Any Distribution
<i>Static FT</i> [Vesely02] (see Sec. C.1)	X	X	X	✓
<i>CFT</i> [Kaiser03] (see Sec. C.2)	X	✓	X	✓
<i>HiP-HOPS</i> [Papadopoulos11] (see Sec. C.3)	X	✓	X	X
<i>Dynamic FT</i> [Manno14b] (see Sec. C.4)	✓	X	✓	✓
<i>Structure Function</i> [Merle14] (see Sec. C.5)	X	✓	X	✓
<i>BDMP</i> [Bouissou07] (see Sec. C.6)	✓	X	✓	✓
<i>SEFT</i> [Kaiser07] (see Sec. C.7)	✓	T:✓; A:X;	✓	only exponential

T: Top model

A: Top model's underlying Analysis model

The approaches displayed in the Table 2.6 can be classified according to their capability of analysing: (1) static configuration and non-repairable basic events; (2) static configuration and repairable basic events; (3) dynamic configuration and non-repairable basic events; and (4) dynamic configuration and repairable basic events.

To create an accurate and maintainable dependability evaluation model, we find necessary the following characteristics:

- Component-based modelling and reuse of components.
- Repair characterization of basic events and components.
- Any failure and repair distribution of basic events and components.

From the literature analysis, there exist many different alternatives to address some of these characteristics. However, to the best of our knowledge there is no approach which integrates all these characteristics. Therefore the integration of the Dynamic Fault Tree paradigm with the Component Fault Trees is deemed an interesting approach, so that it is possible to take the best of both worlds. In Chapter 4 we will introduce the concept of Component Dynamic Fault Trees addressing all the aforementioned characteristics. In Chapter 5 we will add complex repair strategies so that we need to rely on more powerful formalisms.

2.4 Design of Dependable Systems: Trade-Off Between Dependability & Cost

Generally, dependability design decisions and objectives are related to trade-off decisions between system dependability attributes and cost. Dependability requirements often conflict with one another, e.g., safety-availability compromise when a fault leads the system to a safe shut-down in order to prevent it from propagating. The time at which design decisions are taken determines the cost that the design process can incur.

Designing a dependable system within considered requirements requires a process to match and tune combination of architectural components so as to find an optimal solution satisfying design constraints. There are other approaches concentrated on the design of dependable systems under the *correct-by-construction* paradigm. For instance, the approach presented in [Lopatkin11] creates a formal system specification preserving the correctness through gradual refinements of the system design model. However, instead of addressing formal *correct-by-construction* design approaches, we will overview those approaches which are aimed at characterizing at design time the implications of design

decisions on dependability and cost.

More specifically, we group dependable design approaches by looking at how system recovery strategies are implemented. For the design of dependable systems, there exist alternative recovery strategies that add redundancies to the system design in order to avoid single points of failure and thus, provide fault tolerance (cf. Subsection 2.2.1). So far, the explicit replication of hardware and software resources has been successfully applied and it is a feasible solution to recover from failures. Interestingly, in some cases, there exist cost-effective solutions that make the repair possible by reusing already existing hardware resources. Accordingly, we group in Subsection 2.4.1 those approaches that replicate the nominal functionality by aggregating additional hardware resources, i.e., *homogeneous redundancies* and on the other side, in Subsection 2.4.2, we group those approaches which are aimed at reusing hardware components to provide a compatible functionality and reduce hardware costs, i.e., *heterogeneous redundancies*.

2.4.1 Design Approaches using Homogeneous Redundancies

The principal issue addressed by the approaches grouped in this subsection is the evaluation of the effect of design choices (e.g., robustness level of components, redundancy configurations) on dependability and cost.

Methodology for designing distributed control systems by [Cauffriez04]

[Cauffriez13] and [Clarhaut09] focused on designing a dependable system based on a design methodology presented in [Cauffriez04]. The main focus of this methodology relies on the early and systematic characterization of dependability criteria during the system design activities (cf. Figure 2.21).

The approach comprehends three types of architectures: functional, equipment, and operational architectures. As Figure 2.21 describes, the design process is characterized as follows: (1) it starts from the characterization of functional and equipment architectures by addressing functional and dependability criteria; (2) the allocation of the functional architecture onto the equipment architecture is evaluated in relation to dependability; (3) as a result, the operational architecture is produced, which could require reconsid-

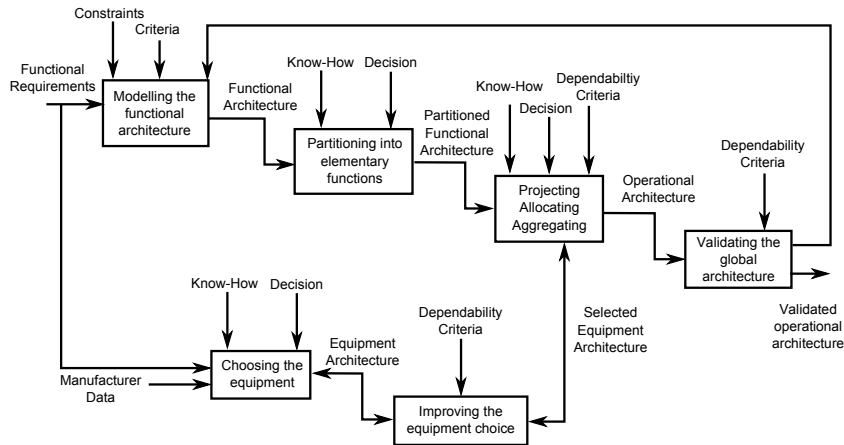


Figure 2.21: Methodology for Designing Distributed Control Systems [Cauffriez04]

ering functional and/or equipment decisions in order to obtain a validated operational architecture with respect to dependability requirements.

Safe-SADT methodology by [Cauffriez13]

The Safe-SADT (Structured Analysis and Design Technique) methodology concentrates on the analysis of repairable architectures by evaluating how the use of alternative hardware components affects system functionality and dependability [Cauffriez13]. To do so, they characterize system-level functions in a top-down manner until lowest level subfunctions are reached. At the bottom layer, failure and repair rates of hardware components permit analysing the performance of the system's top layer (reliability and availability) using Monte Carlo simulations. In this way, a structural function is characterized, which links functions with hardware resources and allows evaluating alternative operational modes by associating different subfunctions to perform the system-level function. The overall design methodology for modelling and analysing alternative architectural design choices has been integrated within a design tool.

Dependable design methodology by [Clarhaut09]

[Clarhaut09] described a design approach overcoming the static-logic limitation of event-based analysis techniques by identifying sequential component-wise contributions to system-level failures. During the design process, a *functional hierarchical tree model*

characterizes dependencies between functions and hardware resources. This model accounts for alternative architectures to perform the modelled control functions. Subsequently, the Improved Multi Fault Tree (IFT) is constructed characterizing sequential failure relationships between Failure Modes (FM) of components and system functions designated as dreaded events.

As Figure 2.22 shows, the structure of the design methodology revolves around the characterization, analysis, and optimization of system architectures so as to adopt optimal design decisions regarding dependability and cost. The IFT determines the dependability level of the overall architecture by weighting the contribution of each component to the system-level failures. Architectural design choices cover active and passive redundancies. The cost associated with each hardware component enables progressing between alternative architectures toward an optimal architecture maximizing dependability and minimizing the cost.

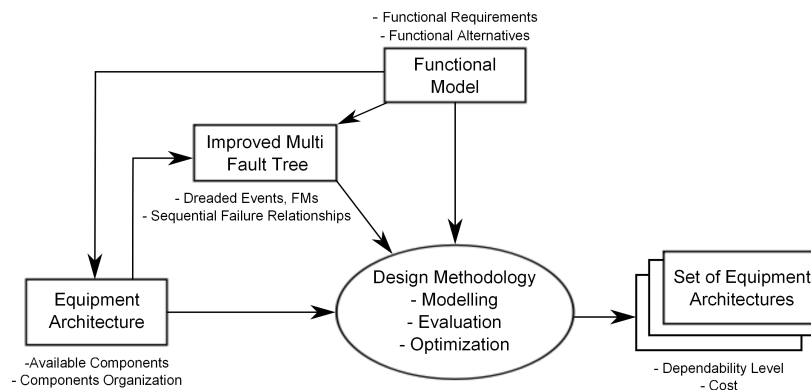


Figure 2.22: Design Approach of [Clarhaut09]

Optimal dependable design architectures by [Adachi11]

[Adachi11] extended the HiP-HOPS approach with recovery strategies to design optimal architectures reducing cost and increasing dependability. The recovery strategies are formally represented using patterns. These patterns characterize the potential to detect, mitigate, and block affecting component failures which are previously identified with HiP-HOPS and analysed by means of Fault Tree Analysis (FTA) and FMEA. Finally, starting from an abstract architecture, recovery strategies are introduced without violating user constraints and an optimization algorithm allows converging through dependability and cost requirements.

Adaptive dependable system design by [Perez14]

More recent approaches continue assessing the influence of homogeneous redundancies from different but closely related points of view. For instance, the work presented in [Perez14], evaluates the influence of the use of adaptive components in the system availability and cost, calculating the availability through GSPN.

All the covered approaches in this subsection aim at increasing system dependability through the explicit replication of nominal components. This design decision implies a cost increase. Consequently, this decision needs to be justified through an exhaustive and adequate analysis of how the system design meets functional and dependability requirements.

2.4.2 Design Approaches using Heterogeneous Redundancies

One of the key properties of the systems which exercise *heterogeneous redundancies* is the ability to successfully accommodate changes in case of failure occurrences. Consequently, the approaches covered in this subsection address dependability issues and adaptation capabilities. Accordingly, they are grouped as *adaptive dependable design approaches*.

Functional alternatives by [Shelton04]

Robust Self-Configuring Embedded Systems (RoSES)⁴ project revolved around the idea to build robust and adaptive embedded systems. [Shelton04] first worked on the concept of *heterogeneous redundancies* by means of *functional alternative* strategies. These strategies allow to compensate for component failures by changing the system functionality. The approach models alternative system configurations and assigns them a relative utility value weighing their contribution to the system performance and dependability. From this model, the overall utility value of the system is calculated which enables the evaluation and comparison of design choices as to where allocate resources for functional alternatives or redundancy.

⁴<http://www.ece.cmu.edu/~koopman/roses/>

Based on each configuration's assigned utility values and system's utility function, alternative configurations are compared. Although this characterization makes it possible to evaluate how component failures affect system utility, the approach has assumptions/limitations to be addressed: (1) there is no calculation of the system's failure probability (dependability analysis techniques are not used); (2) there is no consideration of the influence of health management (fault detection, reconfiguration) and communication mechanisms on the system operation and dependability; and (3) the identification of heterogeneous redundancies is performed case-by-case basis.

Shared redundancy by [Wysocki04]

[Wysocki04] addressed the same design strategy under the *shared redundancy* concept. They concentrated on the reuse of processing units through the strategic distribution of software modules. Consequently, given the failure occurrence of a software component, it is possible to still continue operating through the reconfiguration of communication routes. To evaluate the reliability and safety of the alternative architectures, first a FTA is carried out. This analysis permits extracting minimal combination of events which leads the system to failure (minimal cut-sets). Additionally, this information is used as input for further analysis through Design of Experiments (DOE) to calculate system cost and failure probabilities. Based on the same design concept [Galdun08] analysed the reliability of a networked control system structure using Petri Nets (PN).

The approach concentrates on the reuse of processing units through strategical distribution of software modules among processing units. However, there exist some points worth considering: (1) there is no consideration of the possible compatible functionalities emerging from sensors, actuators and even communication mechanisms; (2) fault detection and reconfiguration mechanisms are assumed ideal; and (3) the dependability analysis models are FTA (without dynamic properties) and Petri nets (flat models limited to exponential failure rates).

ARDEA framework by [Rawashdeh06]

[Rawashdeh06] presented the ARDEA (Automatically Reconfigurable Distributed Embedded Architectures) reconfiguration framework with the goal of designing reconfig-

urable architectures for fault tolerant embedded systems. The approach is based on reconfigurations of processing units to achieve *graceful degradation* and cope with hardware failures. A gracefully degrading system tolerates system failures by providing the same or equivalent functionality with the remaining system components. Dependency graphs are used to model the functional information flow by considering alternative implementations. A centralized system manager uses dependency graphs and a hardware resource list to find a viable mapping of software on the available processing units. It decides when to (un-)schedule software modules by moving object code among available processing units without exceeding processor time and bandwidth.

ARDEA provides an adequate framework for the partial implementation our research ideas. However, (1) they do not perform dependability analysis of the alternative design decisions; (2) heterogeneous redundancies are limited to processing units; (3) heterogeneous redundancies are identified in a ad-hoc manner; and (4) they focus only on centralised reconfiguration implementations.

Implicit redundancies by [Trapp07]

In the MARS project, [Trapp07] proposed a component based modelling and analysis method to exploit *implicit redundancies* so as to react to system failures by reusing hardware resources. They provide methodological support for modelling and gathering system configurations. Moreover, reasonable system configurations are elicited from a set of possible candidates. The adaptive behaviour of the system is modelled based on quality types, which drive system's graceful degradation possibilities.

Each system component operates under different configurations and this is determined by quality attributes which are attached to each component's every I/O port. Configuration activation rules are defined over these ports based on the needed and provided quality attributes (cf. Figure 2.23).

For each component in the system its possible configuration variants are defined. Each port has its own constraints defined as activation preconditions and propagation post-conditions. This characterization determines compatible components, based on quality attributes. As a result, system configurations are extracted based on a explicitly defined adaptation behaviour.

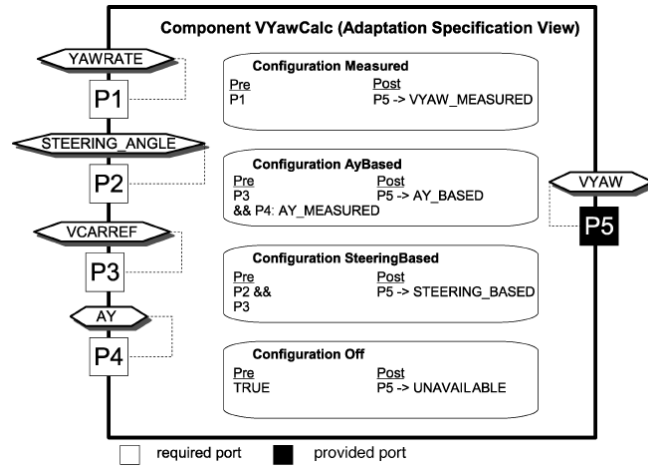


Figure 2.23: Example of an Adaptation Specification View [Adler10b]

From this modelling paradigm (MARS modelling), different analyses have been carried out. In [Adler08], transformations from MARS models into *hybrid-Component Fault Trees* (hybrid-CFTs) were performed in order to calculate configuration probabilities (cf. Subsection 2.3.1). Hybrid-CFTs extend CFTs by using Markov chains models (enclosed in a component) so that it is possible to characterize the repair behaviour of the system. In order to ensure the causality of the reconfiguration sequences and safety-related properties, verification activities have been carried out in [Adler10b]. Last but not least, methodological support for identifying an adaptation model meeting availability-cost trade-off is addressed in [Adler10c].

Despite addressing our similar design goals, this approach has assumptions and there are differences with respect to our methodology: (1) all the approaches within MARS assume fault-free software (ideal fault detection and ideal reconfiguration implementations) and there is no consideration of the influence of the communication on system dependability; (2) the model of adaptation is implemented by a central runtime framework without evaluating the feasibility of distributed reconfiguration implementations; (3) despite considering the use of heterogeneous redundancies, the identification of heterogeneous redundancies is performed in an ad-hoc manner; (4) as for the dependability analysis, the failure/repair characterization of *hybrid-CFTs* models are limited to exponential distributions and its Markov models are limited to characterize basic events.

In the D3H2 methodology (cf. Chapter 3) we focus on addressing all these limitations with connected modelling and analysis activities.

Integrated Modular Avionics

Similar design concepts are addressed in the avionics field. Namely, the Integrated Modular Avionics (IMA) design paradigm defines robust partitioning in on-board avionic systems so that one computing module (line replaceable unit) is able to execute one or more applications of different criticality levels independently. The standardised generic hardware modules forming a network leads to looser coupling between hardware and software applications [Moore01].

SCARLETT project [Bieber09] aims at designing reconfigurable Integrated Modular Avionics architectures in order to mitigate the effect of failures of functional, fault detection and reconfiguration implementations. Once a permanent failure is detected, the reconfiguration supervisor proceeds with the following key activities. Firstly, it manages the modifications given the current configurations and failed module. Secondly, it checks the correctness of the system configuration and the loaded data in the line replaceable unit. The centralized supervisor determines a suitable configuration based on a *reconfiguration graph*, which contains all possible configurations. Reconfiguration policies and real-time and resource constraints, define the set of reachable safe transitions and states. In order to analyse the reconfiguration behaviour when failures occur, a safety model leads to finding the combinations of functional failures [Bieber10].

Based on the same concepts, DIANA project [Engel10] aims at distributing these functionalities. This approach improves the availability of the reconfiguration mechanisms at the expense of relying on a complex, resource consuming communication protocol.

The safety assessment of the reconfigurable Integrated Modular Avionics architectures does consider the influence of the failure of fault detection and reconfiguration implementations on system operation. However, their goal is not to exploit heterogeneous redundancies emerged in massively networked scenarios, instead they are aimed at exploiting replaceable processing units and allocated SW units to perform reconfigurations effectively.

Semantic techniques for dynamic reconfigurations by [Hoftberger13]

Recently, the use of semantic techniques for dynamic reconfigurations in embedded real-time systems has been explored in [Hoftberger13] by means of an ontology and algorithms that enable the runtime adaptation of these systems. The ontology defines expert knowledge about the system structure, relations and interactions between subsystems. When a failure occurs, semantically equivalent services are searched through the ontology. The algorithm determines if a service can be substituted by other services in the system by exploring the ontology to find the required property concepts. Data type, accuracy, and temporal behaviour are compared to check the compatibility of services.

Despite not addressing our target approach for the design process - selection of components, evaluation of its influences on dependability and cost - it does address possible implementation framework for the design concepts treated throughout this dissertation. The performance of the approach relies on the proposed search algorithm, which directly depends on the size of the ontology. For run-time determined reconfigurations this is a critical issue to be addressed.

Fault-tolerant control & fault diagnosis approaches

There have been approaches in the fault-tolerant control and fault diagnosis community aligned with the idea of reusing elements to provide additional functionalities (e.g., see [Blanke11] and references herein). Namely, they focus on identifying analytic redundancies systematically. Proposed approaches in this area evaluate if it is possible to provide the same service with a combination of remaining sensors, i.e., if there exists an alternative analytic equation, which uses different set of variables (resources) to provide the same (or equivalent) service.

The identification of analytic redundancies is based on the structural analysis: relying on detailed mathematical models of the plant (system), system equations, and known and unknown variables are related. If there exists redundant information about the system structure, i.e., if there are more constraints (equations) than variables to be determined, there may exist alternative ways to define a variable (analytical redundancy relations) [Staroswiecki89; Krysanter08]. When dealing with complex systems, detailed models are difficult to obtain. Thus, structural analysis and graph-based analysis emerged to

solve large complex sets of equations [Staroswiecki99]. Analytical redundancy relations are generated from unmatched constraints. Besides, analytical redundancy relations are used as fault detection functions, also known as residuals (e.g., see [Ana10; Svard10]).

Exhaustive characterization and mathematical formulation of complex systems is not trivial and in some cases unfeasible: detailed knowledge about the system is needed to get analytical redundancy relations and therefore, fault-tolerant control of the system. The identification of analytic redundancies through analytical redundancy relations is feasible at subsystem level, but when considering the system as a whole, the complexity of the mathematical formulation increases due to the size of the system and its inner complexity. This is the rationale that led us to adopt a function-based viewpoint with qualitative attributes, instead of the formal mathematical viewpoint (see Chapter 3).

Heterogeneous redundancies are not limited to analytic redundancies. Heterogeneous redundancies include cases in which system variables are not related directly, but they can be derived using system equations and constraints. In massively networked scenarios, systems are comprised of further subsystems - a train is comprised of cars and compartments; or a building is comprised of floors and rooms - which are interconnected using a communication network.

Instead of analysing the consequences on dependability of using alternative configurations, the focus of the fault-tolerant control community has been placed on finding (1) control algorithms able to continue operating in the presence of failures and (2) equations to detect and diagnose failed components.

Diverse redundancies and sensor fusion by [Flammini11]

[Flammini11] introduced a railway security approach which makes use of heterogeneous redundancies. To this end, railway surveillance systems are exploited addressing heterogeneous data sources making use of diverse redundancies and reasoning about heterogeneous data (sensor fusion).

Architecture details are presented integrating the DETECT (DEcision Triggering Event Composer and Tracker) [Flammini09] and SMS (Security Management System) [Flammini10] frameworks. SMS enables to collect the heterogeneous multi-sensor data and store it in a database and DETECT is an scenario-based threat detection approach

(expert system), which provides event correlation mechanisms making possible the generation of alerts based on received inputs, i.e., it is a centralised application of data fusion. Alternative (diverse) sensors and intelligent cameras are used to improve the detection of hazards/attacks, e.g., to identify a contaminated place, Infra-red radiation sensors or Ion Mobility spectroscopy detectors are suggested (analytic redundancy).

Their viewpoint is close to ours, but there are some differences worth mentioning: (1) the use of heterogeneous redundancies is performed in ad-hoc manner providing specific (diverse) solutions to specific problems; (2) the main focus of the approach is on increasing the probability of detection of threat occurrences using diverse implementations; and (3) there is no overall calculation of the failure probability of the system (due to its inner components and their influence).

Safe Software Product Lines by [\[Jean-Pascal13\]](#)

In the project called Safe ReSA (Safe Reusable Safety Analysis and Arguments) [\[Jean-Pascal13\]](#) introduced an approach combining safety engineering and product line engineering disciplines. Software product line engineering focuses on maximizing the reuse through mechanisms to model commonalities and variabilities (feature modelling) [\[Clements01\]](#). The goal of this approach is to apply safety engineering methods to reusable artefacts emerged from product line engineering. To this end, a model-based approach is used to extract safety cases (evidences) covering all the phases: starting from the definition of safety goals until their verification. CFTs are used as a reusable mechanism to analyse cause-effect relations and Safety Concept Trees [\[Domis09a\]](#) to describe how a top-event is refined into a set of safety requirements using combinatorial gates.

Our approach can be linked with software product lines paradigm because we do share the idea of reusing elements: the reuse in software product lines concentrates on creating different systems (product lines) benefiting from the shared properties among systems; we are focused on the reuse of system elements (sensors, controllers, actuators) which already exist in the system. However, our focus relies on reliability engineering: we use redundancies to accommodate changes in case of system failure occurrences and evaluate the influence of alternative design decisions on system dependability and cost (e.g., centralised/distributed reconfiguration implementations or homogeneous/heterogeneous

redundancies). Making an analogy between reliability engineering and software product lines: we analyse system characteristics to find inner variabilities (heterogeneous redundancies) or we add explicit variabilities when necessary (homogeneous redundancies) depending on the dependability and cost constraints. Besides, our approach is limited to networked control systems operating in massively networked scenarios.

2.4.3 Opportunity Analysis

In order to characterize the reviewed approaches within this section, the following design properties have been described in the Table 2.7:

1. Type of recovery strategy.
2. Dependability analysis approach.
3. Cost evaluation.
4. Consideration of the dependability of fault detection (FD), reconfiguration (R) and communication functions.
5. Other tasks, e.g., optimization, verification.

Since the use of *heterogeneous redundancies* requires considering system *dynamics*, the dependability analysis approaches described so far address the temporal behaviour of systems either by linking event-based static-logic approaches with state-based formalisms (e.g., Hybrid-CFT) or by evaluating through approaches which integrate the temporal behaviour explicitly (e.g., Monte Carlo simulations, Dynamic Fault Tree, Petri Nets, Dynamic Bayesian Networks). Moreover, given the *extra* design complexity of the systems which use *heterogeneous redundancies*, the mechanisms which help structuring the analysis and reusing the models are necessary such as hierarchical abstractions or component-based design/analysis paradigms.

To obtain a predictable system design and avoid unexpected failure occurrences, all the approaches assume design-time determined reconfigurations. Nonetheless, it is necessary to go beyond and overcome their underlying assumptions concerning the system's critical functionalities to perform reconfigurations effectively. Namely, among all the reviewed approaches only [Bieber10] and [Adachi11] consider the failure be-

Table 2.7: Approaches and Addressed Design Properties

Works	1	2	3	4	5
[Cauffriez13]	Homogeneous Redundancies	Monte Carlo simulations	HW cost	Not Addressed	Not Addressed
[Clarhaut09]	Homogeneous Redundancies	Improved multi-fault tree	HW cost	Not Addressed	Optimization
[Adachi11]	Homogeneous Redundancies	HiP-HOPS	HW & SW cost	FD, R; Communication not addressed	Optimization
[Perez14]	Homogeneous Redundancies	GSPN	Not Addressed	Not Addressed	Not Addressed
[Shelton04]	Heterogeneous Redundancies	Utility Values	Not Addressed	Assumed Ideal	Optimization
[Wysocki04]; [Galdun08]	Shared Redundancies	Fault Tree, Design of experiments; Petri nets	Maintenance cost	Communication; FD, R Assumed Ideal	Not Addressed
[Rawashdeh06]	Graceful Degradation	Not Addressed	Not Addressed	Not Addressed	Not Addressed
[Hoftberger13]	Reconfigurable Ontology	Not Addressed	Not Addressed	Assumed Ideal	Not Addressed
[Trapp07]	Implicit Redundancy	Hybrid-CFT	HW & SW cost	Not Addressed	Optimization, Verification
[Bieber09] [Engel10]	Reconfigurable IMA	Safety analysis, AltaRica	Not Addressed	FD, R; Communication not addressed	Not Addressed
[Blanke11]	Analytic Redundancy	Not Addressed	Not Addressed	Not Addressed	Residuals, fault diagnosis
[Flammini11]	Diverse Redundancy	(Dynamic) Bayesian Networks	Not Addressed	Not Addressed	Not Addressed
[Jean-Pascal13]	Software Product Lines	Component Fault Trees	Not Addressed	Not Addressed	Not Addressed

haviour of the fault detection and reconfiguration implementations and [Galdun08] addresses the failure the communication network. The evaluation of the possible faulty behaviour of these implementations leads to obtaining an approach which better adheres to real implementations and consequently, more reliable results. Despite not addressing *heterogeneous redundancy* like concepts directly, in [Forster10] an approach called component logic models is presented which does address the faulty behaviour of fault detection implementations.

Shifting from problem specific solutions towards generic fault tolerant design approaches requires systematizing identification, modelling and analysis steps. From our perspective, it is necessary cover the following design activities in order to progress in the design of systems which use heterogeneous redundancies and refine the dependability analysis of these systems:

- Systematic identification of *heterogeneous redundancies* and extraction of system configurations to react in the presence of failures.
- Design of the system architecture to make the use of *heterogeneous redundancies* possible, i.e., fault detection and reconfiguration implementations.
- Evaluation of the system dependability with respect to dependability, adaptivity and cost constraints.

The systematic identification of heterogeneous redundancies and extraction of system configurations calls for an approach which allows identifying systematically existing hardware components able to provide a compatible functionality. To the best of our knowledge, only the work we presented in [Aizpurua12a] works towards this goal. In [Adler10b], authors worked on the systematic extraction of system configurations annotating component by component their adaptive behaviour. During this process they evaluate in a ad-hoc manner if it is possible to provide another configuration variant using alternative hardware components and finally extract system configurations based on inter-component influences. In [Blanke11] a mathematical approach for the systematic identification of analytical redundancies is outlined. It is a sound and consistent approach, but it suffers from scalability issues. The use of this approach within NCSs operating in massively networked scenarios would require too much details concerning the exact mathematical formulation of the system.

The design of the system architecture to make use of *heterogeneous redundancies* requires addressing design decisions regarding the organization of fault detection and reconfiguration implementations, i.e., their distribution and replication. On one hand, when implementing the fault detection function within a networked control system, it is possible to allocate it either on the source PU where the information is produced (e.g., sensor, controller) or in the destination PU, which is the target PU of the source information (e.g., controller, actuator) or in both PUs. On the other hand, when dealing with reconfiguration implementations, its distribution influences the overall dependability and cost of the system (cf. Table 2.8).

Table 2.8: Design Decisions and Influenced Attributes

Design Attribute	Fault Detection		Reconfiguration	
	<i>Source</i>	<i>Destination</i>	<i>Centralised</i>	<i>Distributed</i>
Dependability	Detection at origin, unable to manage communication failures	Detection of wrong value & omission. Prone to common cause failures	Failure proneness: single point of failure	Multiple reconfiguration redundancies
Cost	HW/SW implementation costs	Costly identification of all failures: failure transformation	Single reconfiguration implementation's HW/SW costs	Higher cost: multiple reconfiguration implementations
Complexity	Direct failure handling	Further failure sources	Less communication overhead	Complex communication and resource management

Additionally, when adopting design decisions within the second activity, it is necessary to address adaptivity constraints which also has influence on dependability, e.g., *timeliness constraints*: maximal duration in which the adaptation of one component can be performed [Priesterjahn11b], *dependency constraints*: dependencies between system components, where adapting one component requires further adaptation on other components [Adler10b] or *hardware resource constraints*: limit the use of hardware resources, e.g., processing power, memory [Rawashdeh06].

2.5 Conclusions

Heterogeneous redundancies (or more generally reuse of sensors, controllers and actuators) can take many forms: analytical redundancy, design diversity or fail-safe control algorithms are some well-known examples. To the best of our knowledge, so far this task have been focused on the creative ability of the designer. Furthermore, these approaches have assumed failure-free behaviour of fault detection and/or reconfiguration and/or communication implementations. Besides, previously there have not been an attempt to profit from the physical organization of massively networked scenarios: replicated functions distributed throughout the physical structure of the system. Therefore, so as to integrate these tasks, overcome previous limitations, and evaluate the influence of alternative design decisions on dependability and cost, we have designed a methodology entitled: aDaptive Dependable Design for systems with Homogeneous and Heterogeneous redundancies (D3H2).

Focusing on the dependability analysis of these systems, we have identified the need of an approach which comprehends the characteristics outlined in Subsection 2.3.2. Given that such an approach exists, the analysis of complex, dynamic and repairable systems will become manageable and easier to maintain. In Chapter 4, we will introduce Component Dynamic Fault Tree concept in order to address these characteristics within the D3H2 methodology and in Chapter 5 we will describe a process to model complex repairable systems.

D3H2 Methodology

In order to design adaptive dependable systems systematically and cost-effectively, we propose a design methodology named **aDaptive Dependable Design for Systems with Homogeneous and Heterogeneous redundancies** (D3H2). The methodology integrates the variables implied in the research hypothesis of this thesis: homogeneous/heterogeneous redundancies, fault detection, reconfiguration, and communication.

This chapter is organised as follows:

- As a result of the literature review done in Chapter 2, Section 3.1 introduces the motivation of this chapter.
- Section 3.2 overviews the D3H2 methodology and its main activities.
- Section 3.3 describes in detail the main activities and models for designing a hardware/software architecture systematically including health management strategies.
- Section 3.4 applies the main activities to the *running example* so as to construct a hardware/software architecture.
- Section 3.5 closes this chapter with a discussion of the limitations of the D3H2 methodology.

3.1 Introduction

The design of adaptive dependable systems requires a process to match and tune the adequate combinations of components according to the functional and dependability

requirements. As stated in Chapter 2, the D3H2 methodology emerges from the goal of systematizing all the design steps needed to design complex dependable systems cost-effectively.

Heterogeneous redundancies may exist in diverse systems. However, usually the cost and effort involved in identifying and exploiting which implementations are able to perform further compatible functions is not feasible.

Assuming that potential heterogeneous redundancies have been identified, it is necessary to evaluate quantitatively whether integrating this redundancy is more beneficial - in terms of dependability and cost - than using homogeneous redundancies instead. To evaluate its benefits, firstly it is necessary to create an architecture which makes their use possible. To this end, health management (fault tolerance) mechanisms are required: fault detection and reconfiguration.

When combining all the previous design concepts, the issues that a designer may be interested on covering address: (1) the implementation and distribution of the health management mechanisms; (2) the use of homogeneous or heterogeneous redundancies; (3) trade-off analysis between dependability and cost when selecting alternative architectures (comprised of different interacting elements with their corresponding failure/repair rate and cost).

Repeating this process for different combinations of components (i.e., architectures) can be cumbersome and costly. In consequence, the proposed methodology performs all these tasks systematically.

3.2 Overview of the D3H2 Methodology

The D3H2 methodology focuses on modelling and analysis activities shown in Figure 3.1. The methodology characterizes a system of interest as a set of interacting hardware, software, and communication resources, taking into account their interfaces and provided functionality.

The methodology starts from the characterization of system functions, required resources and the physical location in which these functions are performed. These concepts are

formalised in the *functional model* (cf. Figure 3.1 and Subsection 3.3.1). To this end, the designer has to specify:

- System *functions*.
- List of *resources* in order to meet the system *functions*.
- *Physical location* in which the system *functions* are performed within the system physical structure.

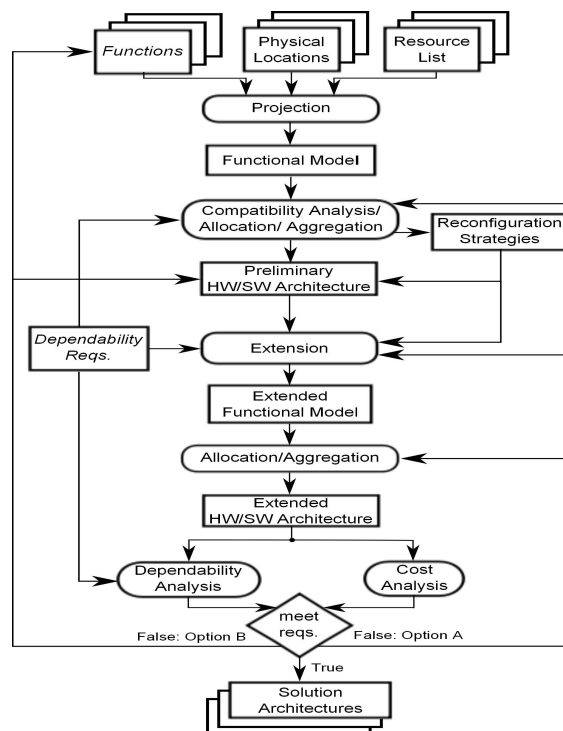


Figure 3.1: D3H2 Methodology [Aizpurua13a]

The *functional model* is obtained from the *projection* of functions onto system resources while considering their *physical location*. This model addresses initial design requirements. To systematize its construction, the Functional Modelling Approach (FMA) has been created (cf. Subsection 3.3.1).

Homogeneous and heterogeneous redundancies are identified as a result of the *compatibility analysis* (cf. Subsection 3.3.2). This activity evaluates if there exist compatibilities in the *functional model*. To take into account these compatibilities, it may be necessary to *aggregate* additional resources and subfunctions and perform the allocation activity for the new elements.

Subsequently, system's *reconfiguration strategies* and the *preliminary HW/SW architecture* are extracted (cf. Subsection 3.3.3).

Before considering this *preliminary HW/SW architecture* for further characterizations and possible implementation, it is necessary to *extend* it with health management functions, i.e., *fault detection* and *reconfiguration*, which make the use of redundancies possible (cf. Subsection 3.3.4). From the *extension* of the *preliminary HW/SW architecture*, the *extended functional model* is constructed.

The *aggregation* and *allocation* activities allow the designer to create an *extended HW/SW architecture* from the *extended functional model*. Subsequently, the *dependability analysis* evaluates the dependability level of the *extended HW/SW architecture* (see Chapter 4 and Chapter 5). Moreover, the *cost analysis* allows adopting trade-off decisions between the used redundancies and incurred cost. Finally, the *extended HW/SW architecture* needs to be evaluated with respect to system requirements to *verify* if the initial requirements are met.

If system requirements are not satisfied there are two options: *Option A* jumps back to a previous activity and repeats the process from there (compatibility analysis, extension, allocation, aggregation). *Option B* drives the design process to the starting point of the design methodology so that design requirements are reconsidered.

The identification of heterogeneous redundancies requires studying all the system functions, resources, and their physical locations early at the design time. Nevertheless, at the expense of relying on a more costly design methodology - rather than simply adding explicit redundant resources where they are necessary - it is expected that the cost savings obtained with heterogeneous redundancies reward the design efforts. The hardware cost savings emerge from limiting the addition of hardware resources (*homogeneous redundancies*) by exploiting already existing hardware resources (*heterogeneous redundancies*). This is something that will be evaluated in Chapter 4 and Chapter 5.

3.3 HW/SW Architecture Design

In Subsection 3.3.1 the Functional Modelling Approach (FMA) is presented. The FMA allows the systematic identification of homogeneous and heterogeneous redundancies

[Aizpurua12a]. Making use of the constructed functional model, in Subsection 3.3.2 and Subsection 3.3.3 the compatibility analysis and reconfiguration strategies are presented. Finally, the Extended Functional Modelling Approach (EFMA) is introduced in Subsection 3.3.4. The EFMA adds fault detection and reconfiguration implementations to the preliminary HW/SW architecture.

3.3.1 Functional Modelling Approach

The overall goal of defining the Functional Modelling Approach (FMA) is the procedural consideration of system functions, resources and the relations between them. The FMA has been designed deliberately to enable the systematic identification of *heterogeneous redundancies* and the extraction of reconfiguration strategies.

The Functional Modelling Approach is characterized in a top-down manner, starting from a set of *high-level functions* tracing down to the necessary resources to perform these functions (cf. Figure 3.2).

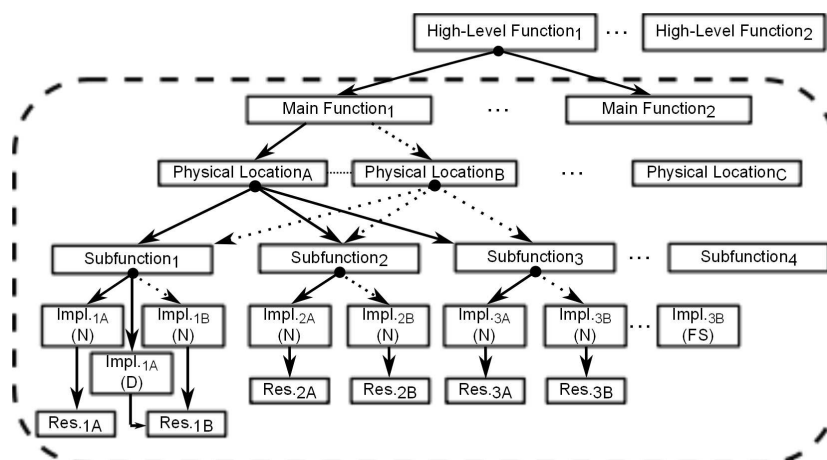


Figure 3.2: Functional Modelling Approach

A high level function (e.g., different train operations: train operating properly, train stopped) is comprised by a set of *Main Functions (MFs)*, e.g., train operating properly = {traction system OK, signalling system OK, braking system OK,...}. These main functions are performed in possibly different Physical Locations (PLs), e.g., a single Air Conditioning Control implementation may span a whole train car or each car compartment in a train car may have its own Air Conditioning Control. In the same way, a

main function consists of a set of *Subfunctions (SFs)*, e.g., input, control and output subfunctions. A subfunction may have multiple implementations and each implementation requires a set of resources that may be shared with other implementations of other subfunctions (e.g., processing units).

Our model focuses on main functions and its sub-levels to limit the scope of the analysis without losing its generality (see the end of this subsection for further discussion). The full characterization of a subfunction’s implementation of a generic main function is specified as follows:

$$\textit{Main Function}.\textit{Physical Location}.\textit{Subfunction}.\textit{Implementation} \quad (3.1)$$

This characterization is comprised of *tokens* which describe the particular *Main Function*, *Physical Location*, *Subfunction* and *Implementation*. These tokens are separated by dots. However, for those tokens which also have dots we surround them with square brackets, e.g., considering the *Physical Location* = Train.Car.Zone:

$$\textit{MainFunction}.[\textit{Train.Car.Zone}].\textit{Subfunction}.\textit{Implementation}.$$

As a result, different tokens are identified straightforwardly (see Example 3.3.1).

To define the physical location of system functions consistently, a physical location map is defined for the physical structure. Figure 3.3 shows the physical location map of an hypothetical train, where each car is comprised of different compartments (Zone_A , Zone_B).

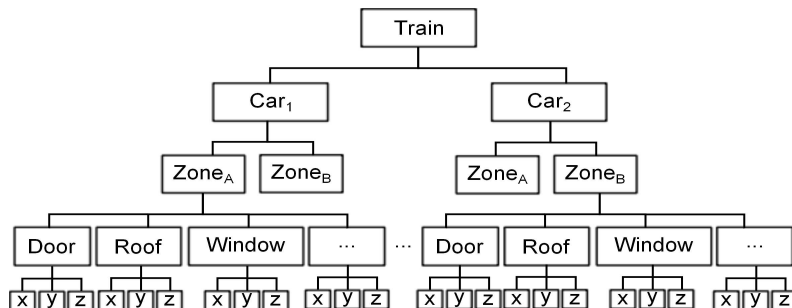


Figure 3.3: Example of a Train Physical Location Map

The physical location of each main function specifies the scope of the main function and its subfunctions, e.g., zone - level: $[Train.Car_1.Zone_A]$; or specific - level: $[Train.Car_2.Zone_B.Door]$.

Example 3.3.1:

The characterization of the Air Conditioning Control main function's temperature measurement subfunction, which is performed within a train car in a specific compartment, will be specified as follows (cf. Section 2.1):

$$AirConditioningControl.[Train.Car_1.Zone_A].TemperatureMeasurement.Sensor_A$$

Where the implementation $Sensor_A$ is comprised of the next set of resources: $Sensor_A = \{Temperature\ sensor\ A, PU_{ACC_A}, SW_{Temp}\}$.

A system *configuration* is defined as follows: a possible realization of the main function comprised of the necessary subfunctions and their underlying implementations (and resources) to perform the main function (cf. Example 3.3.2).

Example 3.3.2:

Considering the Air Conditioning Control main function introduced in Section 2.1 and assuming the train configuration described in Figure 3.3, the nominal configuration for the $Train.Car_1.Zone_A$ will be comprised of the following implementations:

$$AirConditioningControl.[Train.Car_1.Zone_A].TempMeasurement.Sensor_A$$

$$AirConditioningControl.[Train.Car_1.Zone_A].RefTemp.RefButton_A$$

$$AirConditioningControl.[Train.Car_1.Zone_A].TempControlAlgorithm.PID_Control$$

$$AirConditioningControl.[Train.Car_1.Zone_A].Heating.Heater_A$$

where,

$$Sensor_A = \{Temperature\ Sensor\ A, PU_{ACC_A}\};$$

$$RefButton_A = \{Reference\ Temperature\ Button\ A, PU_{ACC_A}\};$$

$$PID_Control = \{PU_{ACC_A}, SW_{PID}, TempMeasurement, RefTemp\};$$

$$Heater_A = \{Heater\ A, PU_{ACC_A}, TempControlAlgorithm\}$$

The *high-level functions* modelled in Figure 3.2 describe the high-level operation of the system. These functions have their own set of main functions and the main functions are comprised of a set of subfunctions. Consequently, if the whole functional model is taken into account, the complexity of the analysis grows up. Accordingly, its manageability is worse and the utility of the Functional Modelling Approach is affected. It is for that reason that we model starting from main functions.

Among the alternative implementations to perform the same subfunction, based on the implementation's resources and provided functionality, we classify nominal, degraded and fail-safe implementations (cf. Figure 3.2). *Nominal (N)* implementations perform the same functions as intended by the initial functional design characteristics. When the nominal implementations are lost due to the failure of some resource, there may be implementations which provide a *Degraded (D)* but acceptable service. *Fail-Safe (FS)* implementations emerge from the need to cope with the severe failure of resources, which could result in hazard situations. In safety-critical systems, fail-safe implementations must be defined to avoid these situations.

Despite the described design methodology concentrates on the design of new systems, it may be customized for the design of already existing systems. Both methodologies differ in the orientation of the construction of the functional model. However, for design purposes, once the functional model is created the same steps apply for both design strategies.

When designing a new system, the orientation of the Functional Modelling Approach focuses from system main functions toward resources (top-down). This design strategy requires planning and understanding completely the system so that an overall picture of the system is obtained. Nonetheless, the drawback of this perspective is that it increases the development time and sometimes not everything is known at the beginning of a project (e.g., physical layout of the system).

On the contrary, when addressing the redesign of an already existing system, a bottom-up first step is needed to obtain a functional model. As it is shown in Figure 3.4, the functional model is constructed by grouping system resources to perform subfunctions and linking them with the main functions they carry out (*synthesis*).

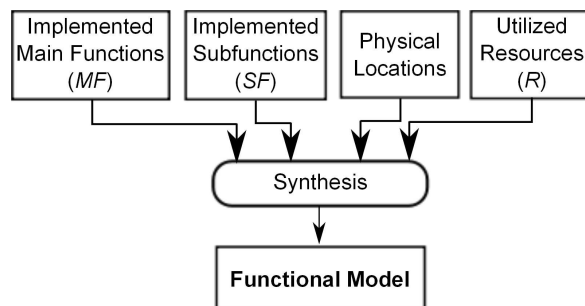


Figure 3.4: Functional Modelling Approach for Existing Systems

When redesigning an existing system, the synthesis of previously designed implementations and functionalities is a troublesome task: taking into account all implementations, subfunctions, and main functions becomes time consuming and prone to errors.

3.3.2 Compatibility Analysis

The objective of the compatibility analysis is the systematic identification of heterogeneous redundancies.

The compatibility analysis allows gathering compatible implementations and identifying heterogeneous redundancies. Two implementations are compatible if they provide the same or similar (but acceptable) result. However, the acceptable results need to be confirmed in a case-by-case basis by the designer. Heterogeneous redundancies are identified based on the *tokens* of the main function: matching subfunctions and compatible physical locations.

There exist two compatibility cases:

- *Natural compatibility* emerges automatically from compatible implementations carrying out the same subfunction in compatible physical locations (cf. Example 3.3.3);
- *Forced compatibility* identifies available I/O implementations located at compatible physical locations, and then evaluates if they may fulfil additional subfunctions with compatible implementations (cf. Example 3.3.5).

Possible *compatible physical locations* are defined as Table 3.1 displays.

Table 3.1: Possible Compatible Physical Locations

Case	Description	Example
1	Subfunctions located at the same physical location	$[\text{Car}_1].\text{Zone}_A \leftrightarrow [\text{Car}_1].\text{Zone}_A$
2	Subfunctions located at adjacent physical locations	$[\text{Car}_1].\text{Zone}_A \leftrightarrow [\text{Car}_1].\text{Zone}_B$
3	Subfunctions located at a physical location that span other subfunction located at more specific physical locations	$[\text{Car}_1].[\text{Zone}_A] \rightarrow [\text{Car}_1].[\text{Zone}_A].\text{Door}$

In the *natural compatibility* case, the compatibility of the physical location depends upon the type of the examined subfunction. For input subfunction implementations performed within compatible physical locations and depending on the input subfunction type itself, the produced outcomes of the implementations are acceptable (cf. Example 3.3.3).

Example 3.3.3:

Considering the temperature measurements in two adjacent compartments:

AirConditioningControl.[Train.Car₁.Zone_A].TemperatureMeasurement.Sensor_A

AirConditioningControl.[Train.Car₁.Zone_B].TemperatureMeasurement.Sensor_B

We identify that the same subfunction's implementations are located at adjacent physical locations ([Train.Car₁.Zone_A], [Train.Car₁.Zone_B]), and the temperature measurements carried out by Sensor_A and Sensor_B could be interchanged in a degraded mode of operation (assuming that the temperature difference in two adjacent compartments is not significant).

However, specific physical locations limit the compatibility. Generally, this is the case of output subfunctions due to their specific actuation space (see Example 3.3.4).

Example 3.3.4:

Considering the specific actuation space of the door manipulation subfunctions' motors located at each doors of a train car (cf. Section 2.1):

DoorStatusControl.[Train.Car₁.Zone_A.Door].DoorManipulation.Motor_A

DoorStatusControl.[Train.Car₁.Zone_B.Door].DoorManipulation.Motor_B

Motor_A can only act in the [Train.Car₁.Zone_A.Door] and it can not manipulate the door located in [Train.Car₁.Zone_B.Door].

Forced compatibility case is analyses available I/O implementations and their physical locations and evaluates if they may fulfil additional subfunctions (cf. Example 3.3.5).

Example 3.3.5:

Consider a train car with the following functionalities (cf. Section 2.1):

DoorStatusControl.[Train.Car₁.Zone_A.Door].DoorClosedDetection.ClosedSensor

VideoSurveillance.[Train.Car₁.Zone_A].VideoInput.Camera

Given that we could add a software functionality to detect the closure of the doors using the camera (e.g., *ClosedCamera* implementation), we consider these implementations compatible:

DoorStatusControl.[Train.Car₁.Zone_A.Door].DoorClosedDetection.ClosedSensor

DoorStatusControl.[Train.Car₁.Zone_A.Door].DoorClosedDetection.ClosedCamera

Using the tokenized characterization of system functionalities (cf. Characterization 3.1), the identification of *redundancies* is simplified. This is performed in a straightforward way by comparing the corresponding tokens of subfunctions and physical locations.

Based on the equivalences between system functions, physical locations and resources, Table 3.2 displays a comparison between the nominal main function configuration and those which use *homogeneous* and *heterogeneous redundancies*.

Table 3.2: Comparison of Redundancies with respect to the Nominal Configuration

Redundancy	Subfunction	Physical Location	Resources	Configuration
Homogeneous	=	=	=	=
Heterogenous	=	≡, =	≡, =	≡

same(=); compatible(≡)

Diverse redundancies (see Subsection 2.2.2) provide the same functionality using an alternative configuration. The difference between diverse and heterogeneous redundancies lies on the design purpose: while heterogeneous redundancies already exist in the system configuration, diverse redundancies are added explicitly to provide the system with implementations which fail in different failure modes and avoid common cause failures. As for the comparison between homogeneous and diverse redundancies, both exercise additional resources, but homogeneous redundancies provide the same function within the same physical location under the same configuration, whereas diverse redundancies provide the same function with a compatible configuration.

The control subfunctions are a special case because they do not depend upon the physical location. They are able to perform the control subfunction provided it receives the corresponding input values of the specific physical location. There may also exist alternative fault-tolerant control subfunction implementations, which are able to cope with input subfunction implementation failures, e.g., open-loop control algorithms.

The implementations identified in the compatibility analysis may be degraded implementations. They are created from a implementation which is a nominal implementation for another main function by reusing resources. We assume that their functionality is acceptable, but they may influence the quality of the provided main function. Accordingly, the validation of the identified heterogeneous redundancies is an activity which will

determine whether the quality of the heterogeneous redundancy implementation is acceptable, e.g., timing requirements of an implementation. These are some challenges to be addressed in our future work to further refine the compatibility analysis (cf. Section 3.5).

3.3.3 Reconfiguration Strategies

To integrate the functional model with heterogeneous and homogeneous redundancies in the D3H2 methodology, *reconfiguration strategies* are defined. Reconfiguration strategies consist of possible system configurations and they describe fault tolerance strategies of the system to recover from system implementation failures.

The existence of compatible implementations lead us to define alternative configurations. These are annotated in a *reconfiguration table* defining all implementations and assigning priorities to each of them (see Example 3.3).

Example 3.3.6:

Table 3.3 displays a hypothetical Air Conditioning Control main function (see Example 3.3.2) with three configuration examples (C_1 , C_2 , C_3), where C_1 refers to the nominal configuration ; C_2 shows a degraded operation reusing a sensor; and the C_3 indicates another degraded operation reusing the reference button.

The hypothetical Air Conditioning Control for Train.Car₁.Zone_A is comprised of 2 heterogeneous redundancy implementations: #1 ↔ #2 and #3 ↔ #4.

The prioritization process for alternative implementations is founded on a metric based on the weighted sum of: (1) level of the degradation of the functionality; (2) failure probability of the implementation; and (3) cost of the configuration. The level of the degradation depends on the relative physical distance (applicable for heterogeneous redundancies emerging from natural compatibilities). This metric does not indicate the final failure probability of the system since it is necessary to extend the system architecture with the necessary health management functions and implementations (cf. Subsection 3.3.4). Besides, in some cases, it is necessary the designer’s knowledge, e.g., when there exist multiple heterogeneous redundancies raised from forced compatibilities. However, it provides an initial idea of the priority of each implementation to perform the subfunction.

Table 3.3: Reconfiguration Table Example

<i>Implementation</i>	Prio	C₁	C₂	C₃	#
AirConditioningControl.[Train.Car ₁ .Zone _A].MeasureTemp.Sensor _A	1	W	F	W	1
AirConditioningControl.[Train.Car ₁ .Zone _A].MeasureTemp.Sensor _B	2		W		2
AirConditioningControl.[Train.Car ₁ .Zone _A].RefTemp.RefButton _A	1	W	W	F	3
AirConditioningControl.[Train.Car ₁ .Zone _A].RefTemp.RefButton _B	2			W	4
AirConditioningControl.[Train.Car ₁ .Zone _A].TempControlAlgorithm.PID	1	W	W	W	5
AirConditioningControl.[Train.Car ₁ .Zone _A].Heating.Heater _A	1	W	W	W	6

W: Working; *F*: Failed; *Prio*: Priority.

Moreover, the reconfiguration strategies enable the direct identification of single points of failure. A single implementation of a subfunction in the reconfiguration table indicates that the subfunction is a single point of failure (e.g., #6 in Table 3.3).

One of the limitations of the studied reconfiguration strategies is the process needed to extract the reconfiguration strategies. That is, the characterization of all the system functions, resources and their physical locations is a laborious task.

3.3.4 Extended Functional Modelling Approach

The main goal of the *Extended Functional Modelling Approach (EFMA)* is to add health management functions and corresponding implementations to the preliminary HW/SW architecture. Namely, it is necessary to add:

- Fault Detection (FD) mechanisms to detect the incorrect operation of an implementation;
- Reconfiguration (R) mechanisms to recover from implementation failures.

The EFMA has been designed with the goal of making it general enough to allow the systematic design and analysis of alternative extended HW/SW architectures. Since fault detection and reconfiguration subfunctions are subfunctions of a given main function, they are also modelled using tokens (cf. Characterization 3.1).

The following design assumptions are adopted when characterizing the health management subfunctions and their implementations [Aizpurua13a]:

- **Fault detection:**

- Each subfunction has an associated fault detection subfunction (FD_SF);
- All the fault detection implementations of the same subfunction use replicas of the same fault detection algorithm;
- The fault detection subfunction is located at the destination processing unit where the information of the source processing unit is used. This decision enables to detect communication (timing and value) failures straightforwardly.

- **Reconfiguration:**

- Each subfunction will have its own reconfiguration subfunction (R_SF), which receives fault detection subfunction's signals and sends reconfiguration signals to subfunction implementations.

- **Fault detection of the reconfiguration:**

- Each reconfiguration implementation will have its own fault detection mechanism (FD_R_SF) implemented in keepalive configuration. Each reconfiguration subfunction implementation sends keepalive signals to all their fault detection function implementations (FD_R_SF) to indicate that it is operating (i.e., it is alive). In the absence of a keepalive signal during a predetermined time slot, the reconfiguration implementation of R_SF is assumed to be failed. When this happens, the reconfiguration's fault detection implementation (FD_R_SF) sends an activation signal to the available reconfiguration implementation (R_SF) with the highest priority.

Instead of including **communication** function as part of the *Extended Functional Modelling Approach*, it is considered as a resource to carry out the characterized subfunctions.

Although we have assumed that the implementation of the fault detection function is allocated on the destination PU, it is possible to allocate it on the (1) source PUs where the original subfunction is carried out, (2) destination PUs where the original subfunction's results is being used, or (3) both. On the one hand, if the fault detection

is allocated on the source PU, it is also necessary to have a mechanism which detects its performance omission (see Subsection 2.2.1) in the destination PU where it is used. On the other hand, the allocation of the fault detection function on the destination PU enables implementing a single fault detection function for the different implementations of the same subfunction (e.g., based on time and value thresholds). The latter case requires taking into account at design-time all possible destination PUs and supplying them with fault detection functionalities.

Concerning the implementation of the reconfiguration function we assume that: (1) all subfunction's PUs have a reconfiguration mechanism which enables them to send/receive data to/from different destinations/sources and (2) additionally, there is one (or multiple) decision PU(s) to manage the reconfigurations according to the subfunction's status. If all reconfiguration decision functions are allocated to the same PU, we end up with a centralised decision PU and it becomes highly sensitive to communication failures. On the other end, if reconfiguration decision functions are distributed throughout the system resources, the management of the reconfiguration decision functions becomes much more complex, but less sensitive to common cause failures (cf. Table 2.8).

Figure 3.5 describes an abstract architecture of the main function i and the health management mechanism of the main function's output subfunction. In this figure overlapped rectangles describe alternative implementations for the same subfunction.

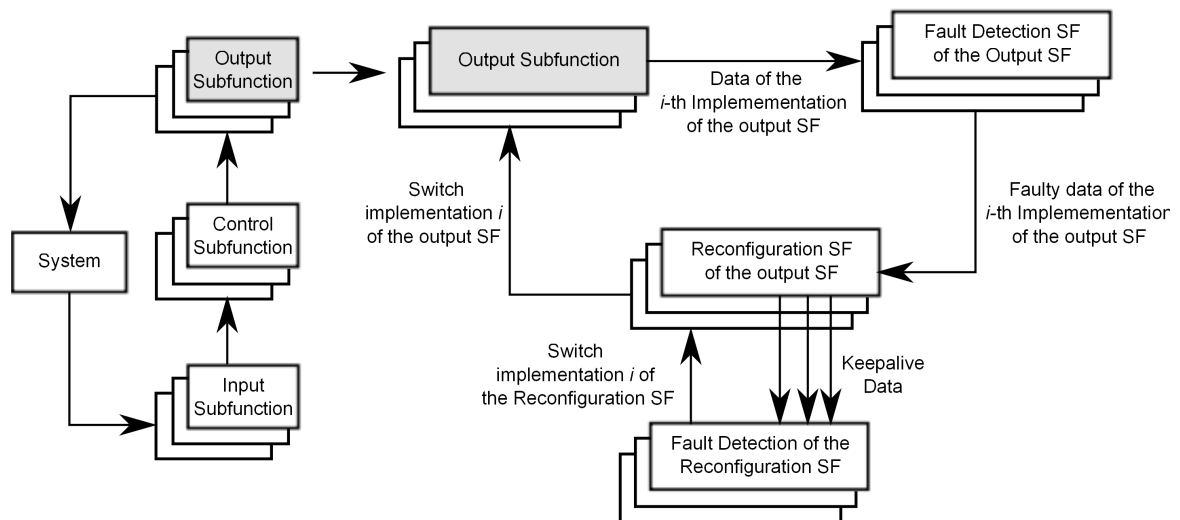


Figure 3.5: Abstract Architecture of the Main Function i and the Health Management Implementation of its Output Subfunction

There is no unique valid solution when allocating resources to fault detection and reconfiguration functions. For instance, when considering the *reconfiguration* function, alternative HW/SW architectures emerge depending on its distribution. This model is general enough to allow for the systematic analysis of alternative HW/SW architectures by means of the *Dependability Evaluation Modelling approach* (cf. Chapter 4 and Chapter 5).

When considering the implementation of the reconfiguration strategies, we assume design-time distribution of alternative configurations. Once reconfiguration strategies are characterized and completed with network addresses of different implementations, the reconfiguration table will be allocated partially in different decision PUs or totally in a unique decision PU to enable the runtime reconfiguration of implementations.

To make the reconfiguration possible, the following needs to be implemented in each PU which has implementations to be reconfigured: a wrapper that ensures the interchangeability between compatible implementations; and a reconfiguration mechanism to redirect its information to different destinations dynamically. Furthermore, the PUs in which the fault detection of the reconfiguration subfunction implementations are allocated require monitoring keepalive signals to control the correct operation of the active reconfiguration implementation.

For these architectures the communication paradigm plays an important role: the communication protocol needs to be able to support the creation/removal of communications dynamically while considering the synchronization of the implementation's states and adjudication of the results. Message oriented publisher/subscriber communication protocols (e.g., Data Distribution Service [Pardo-Castellote03]) address these characteristics: alternative source implementations publish data in a network location and the destination implementations subscribe or unsubscribe to a publisher according to the reconfiguration table. Please refer to the Chapter 6 to read about practical implementation details.

3.4 Results

In order to illustrate how to use the modelling and analysis approaches presented in this chapter, we will apply the D3H2 methodology to the train example described in Section 2.1. From the functions performed in a train car, we will concentrate on three main function examples so as to characterize the different compatibility cases: natural compatibility (Air Conditioning Control) and forced compatibility (Fire Protection Control and Door Status Control).

Natural Compatibility example: Air Conditioning Control

Despite not being a critical function for the safe operation of the train, the Air Conditioning Control function offers a simple, but yet an interesting and intuitive example to demonstrate the possibilities for using heterogeneous redundancies emerging from natural compatibilities (cf. Example 3.3.1).

Functional Model

Let us consider a train with different numbered cars ($\text{Car}_1, \text{Car}_2$) and each car constituted by 2 compartments ($\text{Zone}_A, \text{Zone}_B$) according to the physical location map of the train depicted in Figure 3.3. We assume that independent Air Conditioning Control functions are implemented in each compartment of the train.

As displayed in Table 3.4, the Air Conditioning Control main function implementation for each compartment consists of two input subfunctions: *temperature measurement* and *user reference temperature*; one control subfunction: *air conditioning control algorithm*; and one output subfunction: *heating*. User reference temperature subfunction is constituted by two alternative implementations: reference temperature button (#2, #8) and software defined reference temperature (#3, #9); and also the air conditioning control algorithm contains two different implementations: closed-loop PID control algorithm (#4, #10) and open-loop control algorithm, which only requires temperature reference set-point (#5, #11).

Table 3.4: Functional Model for Air Conditioning Control in Train.Car₁

<i>MF</i>	<i>PL</i>	<i>SF</i>	<i>Type</i>	<i>Implementation</i>	<i>Resources</i>	<i>#</i>
Air Conditioning Control	Train. Car ₁ . Zone _A	Temperature Measurement	I	Sensor _A	Temperature Sensor A, PU _{ACC_A}	1
		User Reference Temperature	I	RefButton _A	Reference Temperature Button A, PU _{ACC_A}	2
		User Reference Temperature	I	RefTempSW _A	SW _{RefTemp} , PU _{ACC_A}	3
		Air Conditioning Control Algorithm	C	PID _A	Temperature Measurement, User Reference Temperature, PU _{ACC_A} , SW _{PID}	4
		Air Conditioning Control Algorithm	C	OL _A	User Reference Temperature, PU _{ACC_A} , SW _{OL}	5
		Heating	O	Heater _A	Air Conditioning Control Algorithm, PU _{ACC_A} , Heater A	6
	Train. Car ₁ . Zone _B	Temperature Measurement	I	Sensor _B	Temperature Sensor B, PU _{ACC_B}	7
		User Reference Temperature	I	RefButton _B	Reference Temperature Button B, PU _{ACC_B}	8
		User Reference Temperature	I	RefTempSW _B	SW _{RefTemp} , PU _{ACC_B}	9
		Air Conditioning Control Algorithm	C	PID _B	Temperature Measurement, User Reference Temperature, PU _{ACC_B} , SW _{PID}	10
		Air Conditioning Control Algorithm	C	OL _B	User Reference Temperature, PU _{ACC_B} , SW _{OL}	11
		Heating	O	Heater _B	Air Conditioning Control Algorithm, PU _{ACC_B} , Heater B	12

Legend: I: Input; C: Control; O: Output; Impl.: implementation

The single implementations of the heating subfunction (#6, #12) indicate that it is a single point of failure for the Air Conditioning Control main function. In these cases, if the main function's requirements are stringent, the functional model points out the need to add an homogeneous redundancy. Since in this case we are not dealing with a safety-critical function, we will assume that it is not necessary to add an homogeneous redundancy (designer's architecture-specific decision).

Compatibility Analysis

Automatically identified heterogeneous redundancies according to the natural compatibility case are:

- It is possible to use the temperature sensor located in contiguous compartments for temperature measurement: #1 \leftrightarrow #7.
- It is possible to use the reference temperature button located contiguous compartments for the user reference temperature: [#2, #3] \leftrightarrow [#8, #9].
- It is possible to reuse the system PUs (PU_{ACC_A} , PU_{ACC_B}) to perform the control functions for both compartments: [#4, #5] \leftrightarrow [#10, #11].

All these implementations are considered compatible because the same subfunction is performed in another compatible physical location (cf. Table 3.1). Therefore, alternative implementations provide a degraded (but acceptable) functionality - see coloured cells in Table 3.5. Possible compatible implementations #9 and #11 were left out for simplicity (cf. Table 3.4).

Extended Functional Model

Once the potential heterogeneous redundancies are selected (cf. Table 3.5 coloured cells), the *extended HW/SW architecture* is created for the Air Conditioning Control main function. To this end, the functional model is extended with health management functions and implementations, and then we allocate resources to the aggregated subfunctions (see Table 3.6).

For the *extended HW/SW architecture* example displayed in Table 3.6, we assumed a centralised reconfiguration decision PU co-allocated with fault detection implementations. This design decision improves the fault containment properties of the health management mechanisms, but also adds a single point of failure.

Please notice how the communication influence is taken into account in the destination implementation. For instance, for the temperature measurement subfunction: (1) implementation #2 requires an activation signal from the reconfiguration implementation

Table 3.5: Preliminary HW/SW Architecture for Air Conditioning Control in Train.Car₁.Zone_A

<i>MF</i>	<i>PL</i>	<i>SF</i>	<i>Type</i>	<i>Implementation</i>	<i>Resources</i>	<i>#</i>
Air Conditioning Control	Train.Car ₁ .Zone _A	Temperature Measurement	I	Sensor _{A1}	Temperature Sensor A, PU _{ACC_A}	1
		Temperature Measurement	I	Sensor _{A2}	Temperature Sensor B, PU _{ACC_B} , Comm	2
		User Reference Temperature	I	RefButton _{A1}	Reference Temperature Button A, PU _{ACC_A}	3
		User Reference Temperature	I	RefButton _{A2}	Reference Temperature Button B, PU _{ACC_B} , Comm	4
		User Reference Temperature	I	RefTempSW _A	SW _{RefTemp} , PU _{ACC_A}	5
		Air Conditioning Control Algorithm	C	PID _{A1}	Temperature Measurement, User Reference Temperature, PU _{ACC_A} , SW _{PID} , Comm	6
		Air Conditioning Control Algorithm	C	PID _{A2}	Temperature Measurement, User Reference Temperature, PU _{ACC_B} , SW _{PID} , Comm	7
		Air Conditioning Control Algorithm	C	OL _A	User Reference Temperature, PU _{ACC_A} , SW _{OL}	8
		Heating	O	Heater _A	Air Conditioning Control Algorithm, PU _{ACC_A} , Heater A	9

#4 which is allocated in a different PU (i.e., PU_{ACC_B} and PU_{ACC_A} respectively); (2) the fault detection implementation #3 needs to monitor the correct performance of implementations #1 and #2, from which the implementation #2 is in a different PU. The same logic applies to the remainder of subfunctions and implementations of the main function.

Reconfiguration Table

Table 3.7 displays the reconfiguration table for the Air Conditioning Control main function implemented in the Train.Car₁.Zone_A.

For simplicity, in Table 3.7 only nominal subfunctions with redundancies are included.

Table 3.6: Extended HW/SW Architecture for the Air Conditioning Control Main Function in Train.Car₁.Zone_A

<i>MF</i>	<i>PL</i>	<i>SF</i>	<i>Type</i>	<i>Implementation</i>	<i>Resources</i>	<i>#</i>
Air Conditioning Control	Train.Car ₁ .Zone _A	Temperature Measurement	I	Sensor _{A1}	Temperature Sensor A, PU _{ACC_A}	1
		Temperature Measurement	I	Sensor _{A2}	Temperature Sensor B, PU _{ACC_B} , Comm	2
		FD_Temp.Meas.	FD	FD_Sensor _A	PU _{ACC_A} , SW _{FD_TM} , Comm	3
		R_Temp.Meas.	R	R_Sensor _A	PU _{ACC_A} , SW _{R_TM}	4
		User Reference Temperature	I	RefButton _{A1}	Reference Temperature Button A, PU _{ACC_A}	5
		User Reference Temperature	I	RefButton _{A2}	Reference Temperature Button B, PU _{ACC_B} , Comm	6
		User Reference Temperature	I	RefTempSW _A	SW _{RefTemp} , PU _{ACC_A}	7
		FD_Ref.Temp.	FD	FD_RefTemp _A	PU _{ACC_A} , SW _{FD_RT} , Comm	8
		R_Ref.Temp. _A	R	R_RefTemp _A	PU _{ACC_A} , SW _{R_RT}	9
		Air Conditioning Control Algorithm	C	PID _{A1}	Temperature Measurement, User Reference Temperature, PU _{ACC_A} , SW _{PID} , Comm	10
		Air Conditioning Control Algorithm	C	PID _{A2}	Temperature Measurement, User Reference Temperature, PU _{ACC_B} , SW _{PID} , Comm	11
		Air Conditioning Control Algorithm	C	OL _A	User Reference Temperature, PU _{ACC_A} , SW _{OL}	12
		FD_ACCA	FD	FD_TempControl _A	PU _{ACC_A} , SW _{FD_TCA} , Comm	13
		R_ACCA	R	R_TempControl _A	PU _{ACC_A} , SW _{R_TCA}	14
		Heating	O	Heater _A	Air Conditioning Control Algorithm, PU _{ACC_A} , Heater A	15

Legend: *FD_X*: Fault Detection of the subfunction X; *R_X*: Reconfiguration of the subfunction X; *Temp.Meas.*: Temperature Measurement (TM); *Ref.Temp.*: user Reference Temperature; *ACCA*: Air Conditioning Control Algorithm

Table 3.7: Reconfiguration Table for the Air Conditioning Control Main Function in Train.Car₁.Zone_A

<i>Implementation</i>	<i>Priority</i>	<i>#</i>
AirConditioningControl.[Train.Car ₁ .Zone _A].TemperatureMeasurement.Sensor _A	1	1
AirConditioningControl.[Train.Car ₁ .Zone _A].TemperatureMeasurement.Sensor _B	2	2
AirConditioningControl.[Train.Car ₁ .Zone _A].UserRefTemp.RefButton _A	1	5
AirConditioningControl.[Train.Car ₁ .Zone _A].UserRefTemp.RefButton _B	2	6
AirConditioningControl.[Train.Car ₁ .Zone _A].UserRefTemp.RefButton_SW _A	3	7
AirConditioningControl.[Train.Car ₁ .Zone _A].TempControlAlgorithm.TCA_PID _A	1	10
AirConditioningControl.[Train.Car ₁ .Zone _A].TempControlAlgorithm.TCA_PID _B	2	11
AirConditioningControl.[Train.Car ₁ .Zone _A].TempControlAlgorithm.TCA_OL _A	3	12

The reconfiguration decision PU needs to know the address of the implementations in the reconfiguration table in order to be signalled for (de)activation purposes and make effective the reconfigurations.

In this case, there is no need to distribute the reconfiguration table to different PUs because all subfunction's reconfiguration implementations are located in the same PU. Therefore, this reconfiguration table will be located at the PU_{ACC_A}.

Forced Compatibility example: Fire Protection Control

In order to illustrate the process for the forced compatibility case, in this subsection we analyse the Fire Protection Control main function (cf. Figure 2.9 and Figure 2.10).

Functional Model

In order to construct the functional model, we will limit the physical location to the Train. Car₁. Zone_A. According to the physical location, we model the functions located at the Zone_A in the train Car₁: Fire Protection Control and Air Conditioning Control. There are other functions located at the same physical level (e.g., Light Control), but

for the sake of clarity we limit the functional models displayed in Table 3.8 to these functions.

As described in the Section 2.1, the Fire Protection Control main function detects the presence of fire using a dedicated fire detector and additionally, passengers signal emergency situations directly using a emergency button located in each compartment of the train. In the presence of fire, the Fire Protection Control algorithm activates the sprinklers located at each compartment of the train car.

Table 3.8: Functional Model for the Functions in Train.Car₁.Zone_A

<i>MF</i>	<i>PL</i>	<i>SF</i>	<i>Type</i>	<i>Implementation</i>	<i>Resources</i>	<i>#</i>
Fire Protection Control	Train. Car ₁ . Zone _A	User Emergency Signal (UES)	I	EmergButton _A	EmergencyButton, PU _{FP}	1
		Fire Detection	I	FireDet _A	Fire Detector, PU _{FP}	2
		Fire Control Algorithm	C	FireControl _A	UserEmergencySignal, FireDetection, SW _{FireControl} , PU _{FP}	3
		Fire Extinction	O	Sprinkler _A	FireControlAlgorithm, PU _{FP} , Sprinkler	4
Air Conditioning Control		Temperature Measurement	I	Sensor _A	Temperature Sensor A, PU _{ACC_A}	5
		User Reference Temperature	I	RefButton _A	Reference Temperature Button A, PU _{ACC_A}	6
		User Reference Temperature	I	RefTempSW _A	SW _{RefTemp} , PU _{ACC_A}	7
		Air Conditioning Control Algorithm	C	PID _A	Temperature Measurement, User Reference Temperature, PU _{ACC_A} , SW _{PID}	8
		Air Conditioning Control Algorithm	C	OL _A	UserReferenceTemperature, PU _{ACC_A} , SW _{OL}	9
		Heating	O	Heater _A	AirConditioningControlAlgorithm, PU _{ACC_A} , Heater	10

Compatibility Analysis

Based on the functional model, the compatibility analysis is performed in order to find compatible implementations existing in the Train.Car₁.Zone_A.

Automatically identified heterogeneous redundancies arising from natural compatibilities are possible: we can use the fire detector located in the contiguous compartment (Train.Car₁.Zone_B) to detect fire in the Train.Car₁.Zone_A. However, we will assume that it is not feasible to replace the fire detection subfunction using the fire detector located in the contiguous compartment due to the degraded quality of the heterogeneous redundancy: the time needed to detect a fire using the contiguous compartment’s smoke sensor is assumed to be too high.

Semi-automatically identified heterogeneous redundancies emerging from forced compatibilities are feasible: it is possible to use a temperature sensor to detect the presence of fire using temperature value thresholds: #5 → #2.

Table 3.9: Preliminary HW/SW Architecture for the Fire Protection Control in Train.Car₁.Zone_A

<i>MF</i>	<i>PL</i>	<i>SF</i>	<i>Type</i>	<i>Implementation</i>	<i>Resources</i>	<i>#</i>
Fire Protection Control	Train.Car ₁ .Zone _A	User Emergency Signal (UES)	I	EmergButton _A	EmergencyButton, PU _{FP}	1
		Fire Detection	I	FireDet _A	Fire Detector, PU _{FP}	2
		Fire Detection	I	Sensor _A	Temperature Sensor A, PU _{ACC_A} , SW _{FireDet} , Communication CAN, Communication ETH, Gateway _{ETH-CAN}	3
		Fire Control Algorithm	C	FireControl _A	UserEmergencySignal, FireDetection, SW _{FireControl} , PU _{FP}	4
		Fire Extinction	O	Sprinkler _A	FireControl, PU _{FP} , Sprinkler	5

Note that the Fire Protection Control and Air Conditioning Control main functions are connected to different communication networks, i.e., CAN and Ethernet respectively (cf. Figure 2.1). Therefore, a gateway device is necessary in order to use alternative communication protocol’s data.

Extended Functional Model

To use these redundancies in massively networked scenarios, it is necessary to complete the *extended HW/SW architecture* with health management and communication mechanisms as Table 3.10 displays.

Table 3.10: Extended HW/SW Architecture for the Fire Protection Control in Train.Car₁.Zone_A

<i>MF</i>	<i>PL</i>	<i>SF</i>	<i>Type</i>	<i>Implementation</i>	<i>Resources</i>	<i>#</i>
Fire Protection Control	Train.Car ₁ .Zone _A	User Emergency Signal (UES)	I	EmergButton _A	EmergencyButton, PU _{FP}	1
		Fire Detection	I	FireDet _A	Fire Detector, PU _{FP}	2
		Fire Detection	I	Sensor _A	Temperature Sensor A, PU _{ACC_A} , SW _{FireDet} , Communication CAN, Communication ETH, Gateway _{ETH-CAN}	3
		FD_FireDetection	FD	FD_FireDet _A	SW _{FD_FireDet} , PU _{FP} , Communication CAN, Communication ETH, Gateway _{ETH-CAN}	4
		R_FireDetection ₁	R	R_FireDet _{A1}	SW _{R_FireDet} , PU _{FP}	5
		R_FireDetection ₂	R	R_FireDet _{A2}	SW _{R_FireDet} , PU _{ACC_A} , Communication CAN, Communication ETH, Gateway _{ETH-CAN}	6
		FD_R_FireDetection	FD_R	FD_R_FireDet _{A1}	SW _{FD_R_FireDet} , PU _{ACC_A} , Communication CAN, Communication ETH, Gateway _{ETH-CAN}	7
		FD_R_FireDetection	FD_R	FD_R_FireDet _{A2}	SW _{FD_R_FireDet} , PU _{FP} , Communication CAN, Communication ETH, Gateway _{ETH-CAN}	8
		Fire Control Algorithm	C	FireControl _A	UserEmergencySignal, FireDetection, SW _{FireControl} , PU _{FP} , Communication CAN, Communication ETH, Gateway _{ETH-CAN}	9
		Fire Extinction	O	Sprinkler _A	FireControlAlgorithm, PU _{FP} , Sprinkler	10

Reconfiguration Table

Table 3.11 displays the reconfiguration table for the Fire Protection Control main function implemented in the Train.Car₁.Zone_A. The reconfiguration table includes the line number (#) of each implementation; and *priority* of each implementation to perform a determined subfunction in a defined physical location.

Table 3.11: Reconfiguration Table of the Fire Protection Main Function in the $\text{Train.Car}_1.\text{Zone}_A$

<i>Implementation</i>	<i>Priority</i>	<i>#</i>
$\text{FireProtectionControl}.[\text{Train.Car}_1.\text{Zone}_A].\text{FireDetection}.\text{FireDet}_A$	1	2
$\text{FireProtectionControl}.[\text{Train.Car}_1.\text{Zone}_A].\text{FireDetection}.\text{Sensor}_A$	2	3

For simplicity, in Table 3.11 only nominal subfunctions with redundancies are included. In this case, reconfiguration implementation is located in PU_{FP} (#5) and PU_{ACC_A} (#6). Therefore, this reconfiguration table will be pre-allocated in both PUs for the reconfiguration of the fire detection subfunction.

Forced Compatibility example: Door Status Control

In this subsection we analyse the Door Status Control main function (cf. Figure 2.3 and Figure 2.4).

Functional Model

In order to construct the functional model, we will limit the physical location to $\text{Train.Car}_1.\text{Zone}_A.\text{Door}$. According to the physical location, we model those functions located at the door of the train car: Door Status Control and Video Surveillance. There exist other functions located at the same physical level (e.g., Passenger Counting System), but for the sake of clarity we limit the functional models displayed in Table 3.12 to these functions.

As described in the Section 2.1, the Door Status Control main function (cf. Figure 2.3 and Figure 2.4) requires different input subfunctions to assure the safe operation of door opening/closing: enable subfunctions (enable door driver, enable door passenger), monitoring subfunctions (door open detection, door closed detection, door velocity, obstacle detection) and command subfunctions (door open command and door close command). These input subfunctions are directed toward the door control algorithm subfunction which determines when and how to close the doors through the door manipulation sub-

function. Video Surveillance main function (cf. Figure 2.5 and Figure 2.6) receives video images (video input subfunction), processes them through the process image control subfunction and finally, if it is the case, it raises an alarm using the lamps and sirens connected to the PU_{Cam} .

Table 3.12: Functional Model for the Functions in the $Train.Car_1.Zone_A.Door$

<i>MF</i>	<i>PL</i>	<i>SF</i>	<i>Type</i>	<i>Implementation</i>	<i>Resources</i>	<i>#</i>
Door Status Control	Train. Car ₁ . Zone _A . Door	Enable Door Driver	I	EnableDriv _{A1}	SW _{TCMS} , PU _{TCMS₁} (simplified)	1
			I	EnableDriv _{A2}	SW _{TCMS} , PU _{TCMS₂} (simplified)	2
		Enable Door Passenger	I	EnablePass _A	Enable Door Driver, PU _{Driver} , EnableButton _{Driver} , Comm.	3
		Door Close Command	I	CloseCommand _A	PU _{Driver} , CloseButton _{Driver}	4
		Door Open Command	I	OpenButton _{Driv.A}	PU _{Driver} , OpenButton _{Driver}	5
			I	OpenButton _{Pass.A}	PU _{DSC_A} , OpenButton _{Passenger}	6
		Door Open Detection	I	OpenSensor _A	PU _{DSC_A} , OpenSensor	7
		Door Closed Detection	I	CloseSensor _A	PU _{DSC_A} , CloseSensor	8
		Door Velocity	I	VelocitySensor _A	PU _{DSC_A} , VelocitySensor	9
		Obstacle Detection	I	ObstacleSensor _A	PU _{DSC_A} , ObstacleSensor	10
		Door Control Algorithm	C	DoorControl _A	Enable Door Passenger, Door Close Command, Door Open Command, Door Closed Detection, Door Open Detection, Door Velocity, Obstacle Detection, PU _{DSC_A} , SW _{CL} , Comm	11
		Door Manipulation	O	Motor _A	Door Control Algorithm, PU _{DSC_A} , Motor	12
Video Surveillance	Train. Car ₁ . Zone _A . Door	Video Input	I	VideoIn _A	Camera, PU _{Cam}	13
		Process Image	C	Surveillance _A	Video Input, SW _{Surveillance} , PU _{Cam}	14
		Alarm	O	Siren _A	Process Image, PU _{Cam} , Lamp, Siren	15

Compatibility Analysis

Based on the functional model, the compatibility analysis is performed in order to find compatible implementations existing in the Door Status Control function.

Automatically identified heterogeneous redundancies arising from natural compatibilities are not feasible because the Door Status Control function is not directly replaceable by other functions located in other places. For instance, it is not feasible to control the status of a door located in Car_1 using the status of the Car_2 door, neither it is feasible to use the status of the door located in a contiguous compartment.

The compatibility analysis points out the following heterogeneous redundancies:

- It is possible to use the camera and its PU_{Cam} with the corresponding intelligent software to identify the position of the doors: door open detection (#7) or door closed detection (#8).
- It is possible to use the camera and its PU_{Cam} with the corresponding intelligent software to calculate the speed of the door (#9).
- It is possible to use the camera and its PU_{Cam} with the corresponding intelligent software to detect obstacles in the door (#10).

After the extraction of all the input and output implementations located at compatible physical locations, it is the designer's work to check if among the suggested list of implementations there exist a feasible compatible implementation.

For the sake of readability we will include heterogeneous redundancies solely for the detection of the door open and close positions as displays the preliminary HW/SW architecture in the Table 3.13 (in Chapter 4 and Chapter 5 when analysing alternative architecture configurations, all possible heterogeneous redundancies are considered).

Extended Functional Model

To use these redundancies in massively networked scenarios, it is necessary to complete the functional model with health management (fault detection and reconfiguration) and

Table 3.13: Preliminary HW/SW Architecture for the Door Status Control in the Train.Car₁.Zone_A.Door

<i>MF</i>	<i>PL</i>	<i>SF</i>	<i>Type</i>	<i>Implementation</i>	<i>Resources</i>	<i>#</i>
Door Status Control	Train.Car ₁ .Zone _A .Door	Enable Door Driver	I	EnableDriv _{A1}	SW _{TCMS} , PU _{TCMS₁} (simplified)	1
			I	EnableDriv _{A2}	SW _{TCMS} , PU _{TCMS₂} (simplified)	2
		Enable Door Passenger	I	EnablePass _A	Enable Door Driver, PU _{Driver} , EnableButton _{Driver} , Comm	3
		Door Close Command	I	CloseCommand _A	PU _{Driver} , CloseButton _{Driver}	4
		Door Open Command	I	OpenButton _{Driv.A}	PU _{Driver} , OpenButton _{Driver}	5
			I	OpenButton _{Pass.A}	PU _{DSC.A} , OpenButton _{Passenger}	6
		Door Open Detection	I	OpenSensor _A	PU _{DSC.A} , OpenSensor	7
		Door Open Detection	I	OpenCamera _A	Camera, PU _{Cam} , SW _{OpenDet} , Comm	8
		Door Closed Detection	I	ClosedSensor _A	PU _{DSC.A} , CloseSensor	9
		Door Closed Detection	I	ClosedCamera _A	Camera, PU _{Cam} , SW _{CloseDet} , Comm	10
		Door Velocity	I	VelocitySensor _A	PU _{DSC.A} , VelocitySensor	11
		Obstacle Detection	I	ObstacleSensor _A	PU _{DSC.A} , ObstacleSensor	12
		Door Control Algorithm	C	DoorControl _A	EDP, DCC, DOC, DCD, DOD, DV, OD, PU _{DSC.A} , SW _{CL} , Comm	13
		Door Manipulation	O	Motor _A	DCA, PU _A , Motor	14

Legend: *EDP*: Enable Door Passenger; *DCC*: Door Closed Command; *DOC*: Door Open Command; *DCD*: Door Closed Detection; *DOD*: Door Open Detection; *DV*: Door Velocity; *OD*: Obstacle Detection; *DCA*: Door Control Algorithm

communication mechanisms and adopt design decisions with respect to the use of heterogeneous redundancies as Table 3.14 displays. Due to the size of the Table 3.14, subsequently we introduce the acronyms used in this table: FD_X: fault detection of the subfunction *X*; R_X: reconfiguration of the subfunction *X*; FD_{R_X}: fault detection of the reconfiguration of the subfunction *X*; EDD: Enable Door Driver; EDP: Enable Door Passenger; DCC: Door Close Command; DOC: Door Open Command;

DOD: Door Open Detection; DCD: Door Closed Detection; DV: Door Velocity; OD: Obstacle Detection; DCA: Door Control Algorithm; and DM: Door Manipulation.

As for the reconfiguration decisions, we will assume a duplicated reconfiguration architecture which is initially centralised, but its replicas are located (distributed) in another PU (PU_{Cam}). This design decision requires monitoring whether the reconfiguration implementations are performing correctly or not. To this end, reconfiguration's fault detection implementations are deployed (#12, #13 - FD_R_DOD; #19, #20 - FD_R_DCD;) so as to monitor the (in)correct performance of the reconfiguration implementations (#10, #11 - R_DOD; #17, #18 - R_DCD) and switch them if necessary.

As for the communication influence we check whether dependent subfunctions are implemented in different PUs. For instance in the door open detection subfunction case (#7) (cf. Table 3.14):

- It has an alternative implementation which requires communication for its activation (#8).
- Since the fault detection of this subfunction (#9) is required to monitor the correct performance of the implementations #7 and #8, and implementation #8 is located in a different PU, implementation #9 will also be influenced by the communication.
- as for the reconfiguration implementation (e.g., R_DOD #10, #11), the implementation #11 will also be influenced by the communication in order to be (re)activated to perform its reconfiguration tasks.
- Reconfiguration's fault detection implementations (e.g., FD_R_DOD #12, #13) will be actively monitoring the correct performance of all the reconfiguration implementations (#10, #11). Therefore, it will be directly influenced by the communication.

Reconfiguration Table

Table 3.15 displays the reconfiguration table for the Door Status Control main function implemented in the $Train.Car_1.Zone_A.Door$.

For simplicity, in Table 3.15 only nominal subfunction with redundancies are included.

Table 3.14: Extended HW/SW Architecture for the Door Status Control in the Train.Car₁.Zone_A.Door

<i>MF</i>	<i>PL</i>	<i>SF</i>	<i>Type</i>	<i>Implementation</i>	<i>Resources</i>	<i>#</i>
Door Status Control	Train. Car ₁ . Zone _A . Door	EDD	I	EnableDriv _{A1}	SW _{TCMS} , PU _{TCMS₁} (simplified)	1
			I	EnableDriv _{A2}	SW _{TCMS} , PU _{TCMS₂} (simplified)	2
		EDP	I	EnablePass _A	EDD, PU _{Driver} , EnableButton _{Driver} , Comm	3
		DCC	I	CloseCommand _A	PU _{Driver} , CloseButton _{Driver}	4
		DOC	I	OpenButton _{Driv.A}	PU _{Driver} , OpenButton _{Driver}	5
			I	OpenButton _{Pass.A}	PU _{DSC_A} , OpenButton _{Passenger}	6
		DOD	I	OpenSensor _A	PU _{DSC_A} , OpenSensor	7
			I	OpenCamera _A	Camera, PU _{Cam} , SW _{OpenDet} , Comm	8
		FD_DOD	FD	FD_OpenDet _A	PU _{DSC_A} , SW _{FD_DOD} , Comm	9
		R_DOD	R	R_OpenDet _{A1}	PU _{DSC_A} , SW _{R_DOD}	10
		R_DOD	R	R_OpenDet _{A2}	PU _{Cam} , SW _{R_DOD} , Comm	11
		FD_R_DOD	FD_R	FD_R_OpenDet _{A1}	PU _{Cam} , SW _{FD_R_DOD} , Comm	12
		FD_R_DOD	FD_R	FD_R_OpenDet _{A2}	PU _{DSC_A} , SW _{FD_R_DOD} , Comm	13
		DCD	I	ClosedSensor _A	PU _{DSC_A} , CloseSensor	14
			I	ClosedCamera _A	Camera, PU _{Cam} , SW _{CloseDet} , Comm	15
		FD_DCD	FD	FD_CloseDet _A	PU _{DSC_A} , SW _{FD_DCD} , Comm	16
		R_DCD	R	R_CloseDet _{A1}	PU _{DSC_A} , SW _{R_DCD}	17
		R_DCD	R	R_CloseDet _{A2}	PU _{Cam} , SW _{R_DCD} , Comm	18
		FD_R_DCD	FD_R	FD_R_CloseDet _{A1}	PU _{Cam} , SW _{FD_R_DCD} , Comm	19
		FD_R_DCD	FD_R	FD_R_CloseDet _{A2}	PU _{DSC_A} , SW _{FD_R_DCD} , Comm	20
		DV	I	VelocitySensor _A	PU _{DSC_A} , VelocitySensor	21
		OD	I	ObstacleSensor _A	PU _{DSC_A} , ObstacleSensor	22
		DCA	C	DoorControl _A	EDP, DCC, DOC, DCD, DOD, DV, OD, PU _{DSC_A} , SW _{CL} , Comm	23
		DM	O	Motor _A	DCA, PU _{DSC_A} , Motor	24

Table 3.15: Reconfiguration Table of the Door Status Control Main Function in the Train.Car₁.Zone_A.Door

<i>Implementation</i>	<i>Priority</i>	<i>#</i>
DoorStatusControl.[Train.Car ₁ .Zone _A .Door].DoorOpenDetection.OpenSensor _A	1	7
DoorStatusControl.[Train.Car ₁ .Zone _A .Door].DoorOpenDetection.OpenCam _A	2	8
DoorStatusControl.[Train.Car ₁ .Zone _A .Door].DoorClosedDetection.ClosedSensor _A	1	14
DoorStatusControl.[Train.Car ₁ .Zone _A .Door].DoorClosedDetection.ClosedCam _A	2	15

In this case, reconfiguration implementations are located in different PUs. Therefore, this reconfiguration table will be located at PU_{DSC_A} and PU_{Cam}.

3.5 Conclusions

In this chapter we have introduced the methodology to design HW/SW architectures systematically. This methodology enables the systematic identification of redundancies and single points of failure. An straightforward extension of the initial HW/SW architecture allows the designer to create the completed *extended HW/SW architecture* which account for designer's decisions with respect to the distribution and implementation of fault detection, reconfiguration and communication functions.

The presented modelling approaches (Functional Modelling Approach and Extended Functional Modelling Approach) enable an straightforward characterization of the system and its subsequent exploitation for redundancy identification and further analyses. However, this process requires studying all the system functions, resources, and their physical locations early at the design-time. At the expenses of relying on a more costly design methodology it is expected that the cost savings obtained with heterogeneous redundancies reward the design efforts (cf. Chapter 4 and Chapter 5).

When using heterogeneous redundancies, the designer needs to be aware of the quality degradation and evaluate whether it is acceptable or not. Validation of the heterogeneous redundancies is not a trivial task. Different architecture-specific requirements subject to real system operation need to be taken into account, such as timeliness, memory and

processing capacity constraints of the processing units. These are some challenges to be addressed in our future work to refine the compatibility analysis (see Chapter 7).

Another limitation of the D3H2 approach is the static nature of reconfiguration table. Although the reconfiguration table can be updated directly to reflect system changes, dynamically updating the reconfiguration table would facilitate its maintenance.

Dependability & Cost Analysis of Non-Repairable Systems

This chapter defines the dependability evaluation algorithm to assess the *extended HW/SW architecture* defined in Chapter 3. This algorithm makes possible the systematic/automatic analysis of the influence of alternative architectural design decisions on dependability.

The chapter is organised into the next sections:

- Section 4.1 introduces the analysis paradigm, states the hypotheses that this chapter assumes and sets the motivation and goals of the chapter.
- Section 4.2 presents the Dependability Evaluation Modelling (DEM) approach for non-repairable systems. The analysis algorithm and adopted implementation techniques are presented.
- Section 4.3 describes the implementation of the simulation-based sensitivity analysis within the Dependability Evaluation Modelling approach.
- Section 4.4 explains the implementation of the uncertainty analysis in order to deal with the lack of exact failure-related data information.
- Section 4.5 defines the assumptions and decisions adopted to perform the cost analysis of the system.
- Section 4.6 applies the Dependability Evaluation Modelling approach and sensitivity, uncertainty and cost analyses to the *running example* of this dissertation.
- Finally, Section 4.7 sums up the conclusions of this chapter.

4.1 Introduction

The *extended HW/SW architecture* is comprised of many different design decisions: (1) selection of the type and number of redundancy strategies (homogeneous, heterogeneous); (2) selection of the most adequate reconfiguration scheme (centralised, distributed); (3) selection of the number and type of PUs (with respect to their reliability and cost parameters); or (4) allocation of software functions into the different PUs.

The combination of different design decisions produces different results with respect to dependability and cost. Therefore, there is room to optimize design decisions so as to improve dependability and reduce system cost. The goal of the DEM is to analyse the dependability level of the *extended HW/SW architecture* - which contains any of the previously mentioned design decisions.

In the scientific literature (cf. Chapter 2) there have been approaches implementing the systematic/automatic transformation from design models to dependability analysis models (see Subsection 2.3.1 - *Model-based Transformational Approaches*). Besides, there exists dependability-specific solutions which directly evaluate the influence of architectural design decisions on system's dependability and cost (see Subsection 2.4.1). However, to the best of our knowledge, there are no approaches which analyse the influence of heterogeneous redundancy schemes including the failure behaviour of fault detection, communication and alternative reconfiguration strategies (see Subsection 2.4.2).

To perform the systematic dependability assessment of the *extended HW/SW architecture*, the following assumptions are adopted:

- Fixed architectural design decisions with respect to health management implementations and their allocations (cf. Chapter 3):
 - Fault detection located at the destination implementation.
 - Fault detection of the reconfiguration subfunction implemented as heartbeat or keepalive implementation.
 - Each system subfunction has its own reconfiguration subfunction, which may be centralised or distributed.
- Resource failures are non-repairable.

Illustration of the Problem Addressed by the Chapter

Extended HW/SW architectures are characterized by different design characteristics and failure influences: different reconfiguration sequences, priorities, functional dependencies or common cause failures are some examples of these characteristics. The systematic dependability assessment of the *extended HW/SW architecture* requires taking into account all the possible situations in which the (complex) system is unable to continue performing its design function.

The complexity that emerges from dependencies and influencing hardware, software and communication resources leads to compromising the maintainability (readability, traceability) of the dependability analysis model of the *extended HW/SW architecture*. Therefore, component-based modelling mechanisms [Crnkovic03] are deemed a necessary design instrument to deal with the size and complexities of the *extended HW/SW architecture*.

The design-related research questions that this chapter aims to answer are:

- Which is the influence on dependability and cost of using different reconfiguration and redundancy strategies?
- Which is the contribution of a given component on the system failure?
- Can we assume the ideal performance of health management and communication implementations?

The analysis of centralised and distributed reconfiguration strategies in itself does not pose new challenges. However, the combination of alternative reconfiguration strategies with homogeneous and/or heterogeneous redundancies (redundancy strategies) sets new issues to be analysed:

- When using heterogeneous redundancies, which is the best trade-off in reconfiguration strategy with respect to the cost of the system and its dependability?

Linked with the previous design issue, we will perform importance measurements so as to evaluate quantitatively the influence of homogeneous and heterogeneous redundancies (and related design decisions) on system failure.

So far, design approaches which have considered the use of heterogeneous redundancies have assumed the ideal performance of fault detection and reconfiguration functions (cf. Chapter 2). In this chapter, we aim to evaluate the validity of this assumption through different dependability analyses.

4.2 Dependability Evaluation Modelling Approach

To evaluate the dependability of the *extended HW/SW architectures* systematically and intuitively, we have defined a Dependability Evaluation Modelling (DEM) approach [Aizpurua14].

4.2.1 Concepts and Notation

The objective of the DEM approach is the generic, systematic and complete failure modelling of *extended HW/SW architectures* to evaluate their dependability.

The **failure model** of the *extended HW/SW architecture* includes the following *failure modes*: fault detection implementations fail (FD_SF, FD_R_SF) in *Omission* (O) when it does not detect a failure when it occurs and fail in *False Positive* (FP) when it detects a failure when it does not exist; the reconfiguration implementation fails in *omission* when it fails to reconfigure a faulty implementation; and failure of subfunction's implementations cover value and timing failures. Figure 4.1 shows the failure model of the *extended HW/SW architecture*.

All possible failures of all system subfunction implementations (SF, FD_SF, R_SF, FD_R_SF) are defined at the implementation level ($[MF].[PL].[SF].[Impl]$ *Failure*) according to the failure characteristics of the implementation's resources. Based on the combination of implementation-level failures, subfunction-level failures are defined systematically ($[MF].[PL].[SF]$ *Failure*).

Table 4.1 defines the notations of the failure events and working events according to their subfunction and failure modes. For clarity, in subsequent characterizations we omit the common part ($[MF].[PL]$).

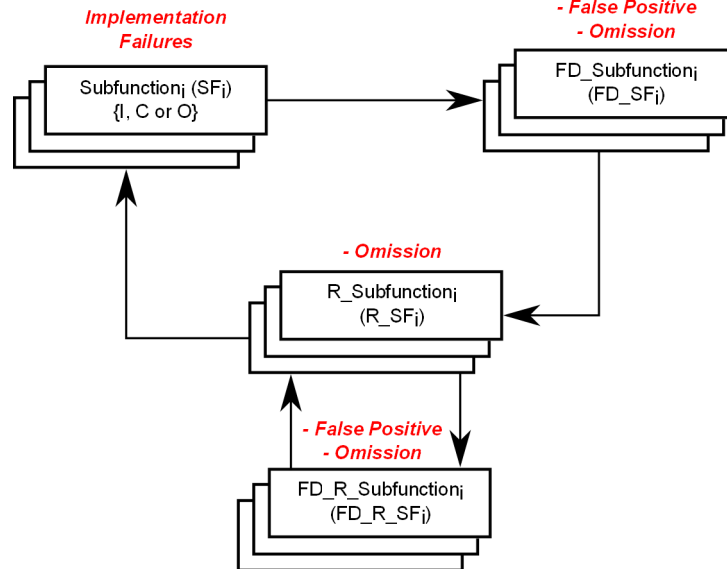


Figure 4.1: Extended HW/SW Architecture's Failure Model

Table 4.1: Notation of Failure and Working Events

<i>Notation</i>	<i>Failure Logic</i>	<i>Notation</i>	<i>Failure/Working Logic</i>
\mathcal{F}_X	X failure	\mathcal{W}_X	X working
\mathcal{F}_{SF}	[SF] failure	\mathcal{W}_{SF_i}	[SF].[Impl _i] working = NOT (\mathcal{F}_{SF_i})
\mathcal{F}_{SF_i}	[SF].[Impl _i] failure	\mathcal{F}_R	[R_SF] failure
\mathcal{F}_{FD}	[FD_SF] failure	$\mathcal{F}_{R_i O}$	[R_SF].[Impl _i] omission
$\mathcal{F}_{FD FP}$	[FD_SF] false positive	$\mathcal{F}_{FD_R_i FP}$	[FD_{[R_SF].[Impl _i]}] false positive
\mathcal{F}_{FD_i}	[FD_SF].[Impl _i] failure	$\mathcal{F}_{FD_R_i O}$	[FD_{[R_SF].[Impl _i]}] omission
$\mathcal{F}_{FD_i O}$	[FD_SF].[Impl _i] omission	$\mathcal{F}_{R_i O/FP}$	[R_SF].[Impl _i] omission or FP = OR ($\mathcal{F}_{R_i O}$, $\mathcal{F}_{FD_R_i FP}$)
$\mathcal{F}_{SF_i FP}$	[SF].[Impl _i] failure or FP = OR (\mathcal{F}_{SF_i} , $\mathcal{F}_{FD FP}$)		

The stochastic failure characterization of each resource is characterized by sampling randomly the failure times according to their Cumulative probability Distribution Functions (CDFs) along the system lifetime. The methodology supports any CDFs, but for the sake of simplicity without losing the generality of the approach, in subsequent probabilistic characterizations exponential failure distributions are assumed.

Therefore, the failure characterization of system resources is defined according to their failure rates (λ_{Res}). The failure characterization of a SF's i -th implementation ($[SF]$. $[Imp_i]$ Failure) comprised of N resources is specified as follows:

$$\mathcal{F}_{\text{SF}_i} = \mathbf{OR}(\lambda_{\text{Res}_1}, \lambda_{\text{Res}_2}, \dots, \lambda_{\text{Res}_N}) \quad (4.1)$$

The same equation holds for the failure characterizations of the omission failures of: FD_SF ($\mathcal{F}_{\text{FD}_i \text{O}}$), R_SF ($\mathcal{F}_{\text{R}_i \text{O}}$), and FD_R_SF ($\mathcal{F}_{\text{FD}_i \text{R}_i \text{O}}$) implementations. Accordingly, the false positive failures of fault detection implementations ($\mathcal{F}_{\text{FD}_{\text{FP}}}$ and $\mathcal{F}_{\text{FD}_i \text{R}_i \text{FP}}$) will be characterized with their characterizing failure distribution and corresponding parameters (e.g., exponential distribution with $\lambda_{\text{FD}_{\text{FP}}}$ and $\lambda_{\text{FD}_i \text{R}_i \text{FP}}$ values).

4.2.2 Analysis Algorithm

The DEM approach defines an algorithm that evaluates the dynamic failure behaviour of systems which use fault detection and reconfiguration implementations while covering all possible failure situations for the specified *extended HW/SW architectures*. It allows to evaluate systematically the consequence of design decisions on system dependability (see Section 4.1). Resulting equations characterize the failure of such systems compositionally so that the failure logic is kept clear for complex systems.

To this end, the DEM approach characterizes combinations of subfunction's implementation failures that prevent the *extended HW/SW architecture* from performing its intended subfunction⁵. The SF will fail (\mathcal{F}_{SF}) when all implementations have failed ($\mathcal{F}_{\text{All Impl.}}$), an implementation fails and reconfiguration does not happen (failure unresolved, $\mathcal{F}_{\text{Unresolved}}$), or its input dependencies have failed ($\mathcal{F}_{\text{Dependencies}}$):

$$\mathcal{F}_{\text{SF}} = \mathbf{OR}(\mathcal{F}_{\text{All Impl.}}, \mathcal{F}_{\text{Unresolved}}, \mathcal{F}_{\text{Dependencies}}) \quad (4.2)$$

Assuming that we have N_{SF} implementations of the subfunction, the $\mathcal{F}_{\text{All Impl.}}$ event happens when each implementation fails or is detected as failed:

⁵Since the failure of any subfunction necessary for a main function provokes the immediate failure of a main function, from this point onwards we will only consider the failure of a subfunction.

$$\mathcal{F}_{\text{All Impl.}} = \mathbf{AND}(\mathcal{F}_{\text{SF}_1 \text{ FP}}, \dots, \mathcal{F}_{\text{SF}_{N_{SF}} \text{ FP}}) \quad (4.3)$$

The failure unresolved ($\mathcal{F}_{\text{Unresolved}}$) occurs when the working implementation fails and either the fault is not detected (failure undetected) or the reconfiguration itself fails (reconfiguration failed). For each implementation there are different failure unresolved events ($\mathcal{F}_{\text{Unr. Imp}_i}$) because each implementation may have different failure probabilities, however, note that the last implementation's failure cannot be solved (non-repairability assumption):

$$\mathcal{F}_{\text{Unresolved}} = \mathbf{OR}(\mathcal{F}_{\text{Unr. Imp}_1}, \dots, \mathcal{F}_{\text{Unr. Imp}_{N_{SF}-1}}) \quad (4.4)$$

To define the failure unresolved event of the i -th implementation of the generic subfunction SF ($\mathcal{F}_{\text{Unr. Imp}_i}$), let us introduce two new events. The first event occurs when the i -th implementation of the subfunction fails and the reconfiguration has failed but after successfully reconfiguring previous $i-1$ implementations (reconfiguration sequence failure, $\mathcal{F}_{\text{R Seq}_i}$). Assuming $\mathcal{F}_{\text{SF}_{1..i-1} \text{ FP}} = \mathbf{AND}(\mathcal{F}_{\text{SF}_1 \text{ FP}}, \dots, \mathcal{F}_{\text{SF}_{i-1} \text{ FP}})$ indicates the failure or false positive from 1 to $i-1$ implementations:

$$\mathcal{F}_{\text{R Seq}_i} = \mathbf{PAND}(\mathcal{F}_{\text{SF}_{1..i-1} \text{ FP}}, \mathcal{F}_{\text{R}}, \mathcal{F}_{\text{SF}_i \text{ FP}}) \quad (4.5)$$

The second event occurs when the i -th implementation of the SF fails and the fault detection of the SF has failed but after detecting correctly previous $i-1$ implementation failures (fault detection sequence failure, $\mathcal{F}_{\text{FD Seq}_i}$). Note that fault detection's false positive and omission failures are mutually exclusive and therefore the false positive does not influence $\mathcal{F}_{\text{FD Seq}_i}$:

$$\mathcal{F}_{\text{FD Seq}_i} = \mathbf{PAND}(\mathcal{F}_{\text{SF}_{1..i-1}}, \mathcal{F}_{\text{FD}}, \mathcal{F}_{\text{SF}_i}) \quad (4.6)$$

Due to the characterization of time-ordered failures, Equations 4.5 and 4.6 cannot be further simplified. Accordingly, i -th implementation's failure unresolved event ($\mathcal{F}_{\text{Unr. Imp}_i}$) occurs when either the fault detection sequence ($\mathcal{F}_{\text{FD Seq}_i}$) fails or the reconfiguration sequence ($\mathcal{F}_{\text{R Seq}_i}$) fails:

$$\mathcal{F}_{\text{Unr. Imp}_i} = \mathbf{OR}(\mathcal{F}_{\text{FD Seq}_i}, \mathcal{F}_{\text{R Seq}_i}) \quad (4.7)$$

Dependencies address Input (I) and Control (C) subfunctions influence on control and Output (O) subfunctions respectively. Control subfunction failure impacts directly the output subfunction failure (C→O); and the influence of input subfunction on control subfunction depends if the system's control configuration is operating in closed loop (C_CL) or open loop (C_OL):

$$\mathcal{F}_{\text{Dependencies}} = \mathbf{OR}(\mathcal{F}_{\text{Dep. C_CL}}, \mathcal{F}_{\text{Dep. C_OL}}) \quad (4.8)$$

Assuming that $\mathcal{W}_{\text{C_X}} = \mathbf{OR}(\mathcal{W}_{\text{C_X}_1}, \dots, \mathcal{W}_{\text{C_X}_{N_{\mathcal{W}}}})$ means that any $N_{\mathcal{W}}$ implementations of the C_X subfunction are working (where $\text{X} = \{\text{CL}, \text{OL}\}$), Equations in 4.9 describe the different input subfunctions that affect each control configuration (I_CL→C_CL, I_OL→C_OL). $\mathcal{F}_{\text{Dep. C_OL}}$ may not happen because the open loop control generally does not have input dependencies:

$$\mathcal{F}_{\text{Dep. C_CL}} = \mathbf{AND}(\mathcal{W}_{\text{C_CL}}, \mathcal{F}_{\text{I_CL}}) \quad \mathcal{F}_{\text{Dep. C_OL}} = \mathbf{AND}(\mathcal{W}_{\text{C_OL}}, \mathcal{F}_{\text{I_OL}}) \quad (4.9)$$

The reconfiguration failure is a special case among subfunctions and therefore \mathcal{F}_{R} is developed like Equation 4.2, except that there are no additional dependencies:

$$\mathcal{F}_{\text{R}} = \mathbf{OR}(\mathcal{F}_{\text{All R Impl.}}, \mathcal{F}_{\text{R Unresolved}}) \quad (4.10)$$

$\mathcal{F}_{\text{All R Impl.}}$ indicates the failure of all reconfiguration implementations and $\mathcal{F}_{\text{R Unresolved}}$ designates the reconfiguration's failure unresolved condition. Assuming M reconfiguration implementations:

$$\mathcal{F}_{\text{All R Impl.}} = \mathbf{AND}(\mathcal{F}_{\text{R}_1 \text{ O/FP}}, \dots, \mathcal{F}_{\text{R}_M \text{ O/FP}}) \quad (4.11)$$

Although the system may operate correctly when a false positive occurs, it has to as-

sume that the information provided by the fault detection is correct, since there is no mechanism to detect the incorrect operation of fault detection.

$\mathcal{F}_{\text{R Unresolved}}$ happens when all M implementations of FD_R_SF fail in omission simultaneously and it is a direct consequence of our design choice: all reconfiguration's fault detection implementations (FD_R_SF) are active and homogeneous redundancies (heartbeat implementations):

$$\mathcal{F}_{\text{R Unresolved}} = \mathbf{AND}(\mathcal{F}_{\text{FD_R}_1 \text{ O}}, \dots, \mathcal{F}_{\text{FD_R}_M \text{ O}}) \quad (4.12)$$

The fault detection failure \mathcal{F}_{FD} is also a special case among subfunctions. It depends on the operation of the destination subfunction (SF_{DEST}), because the FD implementation is located at the same PU. Hence, $\mathcal{F}_{\text{SF_DEST}}$ influences directly \mathcal{F}_{FD} . We assume that the change of destination SF's implementation activates the corresponding FD implementation and the previous one is deactivated. Equation 4.13 describes the FD_SF failure case when FD_SF has K implementations:

$$\mathcal{F}_{\text{FD}} = \mathbf{OR}(\mathcal{F}_{\text{FD_Dest Seq}_1}, \dots, \mathcal{F}_{\text{FD_Dest Seq}_K}) \quad (4.13)$$

As for the i -th fault detection implementation's failure sequence ($\mathcal{F}_{\text{FD_Dest Seq}_i}$), it expresses the following event: from 1 to $i-1$ destination SF's implementations have failed and reconfigured correctly ($\mathcal{F}_{\text{SF_DEST}_{1..i-1}}$), and then either the i -th fault detection or destination SF's implementation fails:

$$\mathcal{F}_{\text{FD_Dest Seq}_i} = \mathbf{PAND}(\mathcal{F}_{\text{SF_DEST}_{1..i-1}}, \mathbf{OR}(\mathcal{F}_{\text{SF_DEST}_i}, \mathcal{F}_{\text{FD}_i \text{ O}})) \quad (4.14)$$

To avoid creating loops when evaluating system's dependability, we have considered that the fault detection implementation's failure is governed by the destination subfunction's implementations failure without considering its input dependencies (cf. Equation 4.14). If destination subfunction's dependencies are taken into account they will create logical loops. Therefore, the influence of dependencies is taken into account at the "top" subfunction's failure level (cf. Equation 4.2). At this level, if any dependent subfunction fails, it leads directly to the failure occurrence of the subfunction.

4.2.3 Analysis of the State of the Art Approaches

In order to implement the equations of the DEM approach, existing dynamic and compositional fault-tree-like paradigms have been analysed (cf. Table 4.2) looking for the following characteristic:

- (1) Component based characterization: embed the failure logic of a set of related events or components and (re)use it where needed instead of characterizing the system failure behaviour in a single flat model.
- (2) Dynamic gates: capture the system failure logic accounting for time-ordered events.
- (3) Support for any probability density function.
- (4) Possibility of modelling repeated basic events.
- (5) Possibility of modelling repeated subsystems or components.
- (6) NOT gates: address the influence of functional events.

Table 4.2: Approach and Characteristics

Approach	(1)	(2)	(3)	(4)	(5)	(6)
Static FT [Vesely02]	X	X	✓	✓	X	✓
Component FT [Kaiser03]	✓	X	✓	✓	X	✓
DFT - Galileo [Dugan92]	X	✓	X	✓	X	X
DFT - RAATS [Manno14b]	X	✓	✓	✓	X	X
DFT - DFTCalc [Arnold13]	T: X; A: ✓	✓	X	✓	X	X
DFT - Radyban [Montani08]	T: X; A: ✓	✓	✓	✓	X	X
DFT - GFT [Raiteri11]	T: ✓; A: X	✓	✓	✓	✓	X
BDMP [Bouissou07]	X	✓	X	✓	✓	X
SEFT [Kaiser07]	T: ✓; A: X	✓	X	✓	✓	✓
HiP-HOPS [Papadopoulos11]	✓	X	X	✓	✓	✓

T: Top model

A: Top model's underlying Analysis model

The integration of static fault trees and compositional characterization is not new [Kaiser03; TU Kaiserslautern09; Adler08]: Component Fault Trees addressed this concept prominently. Among the DFT approaches there exist alternatives to model systems with any failure probability. To this end, simulation-based approaches are used (e.g., RAATSS [Manno14c], Radyban [Montani08]) due to their possibility of approximating such characteristics.

As for the compositional characterization, the Generalized Fault Tree (GFT) (integration of parametric and repairable dynamic fault trees [Bobbio04; Codetta-Raiteri05]) approach is the only one which has worked towards this goal [Raiteri11]. There exist some approaches which model the failure behaviour of a system with a user friendly (compositional) formalisms (top model), but they perform the statistical calculation using a less intuitive (flat) underlying formalism (analysis model). The drawback of the GFT approach relies in the analysis of its underlying formalism (Stochastic Well-Formed nets [Chiola93a]) which is a flat state-based system model which also suffers from state-explosion issues. Besides, the compositional (parametric) viewpoint for this approach is in folding repeated events and symmetric subsystems (see Figure 2.18), but not embedding the same logic in a component and reusing in the same model where deemed necessary as done in Component Fault Trees (see Figure C.2). HiP-HOPS also accounts for the concept of Component Fault Trees using annotations [Papadopoulos11]. Annotated components (which can be seen as Component Fault Trees) are parsed to create the Fault Tree of the system. Despite it has been extended for the extraction of the cut sequence sets, the quantitative solution of dynamic models is not an integrated approach within HiP-HOPS.

To the best of our knowledge, there is no approach which addresses explicitly the integration of Dynamic Fault Trees and component oriented characterization while addressing any failure distributions. To address these characteristics, simulation-based analysis techniques provide adequate analysis mechanisms at the expenses of relying on a increased computation time.

4.2.4 Implementation: Component Dynamic Fault Trees

Addressing all these characteristics, the *concept* of Component Dynamic Fault Tree (CDFT) is defined borrowing the definition of

original Component Fault Trees introduced in [Kaiser03]:

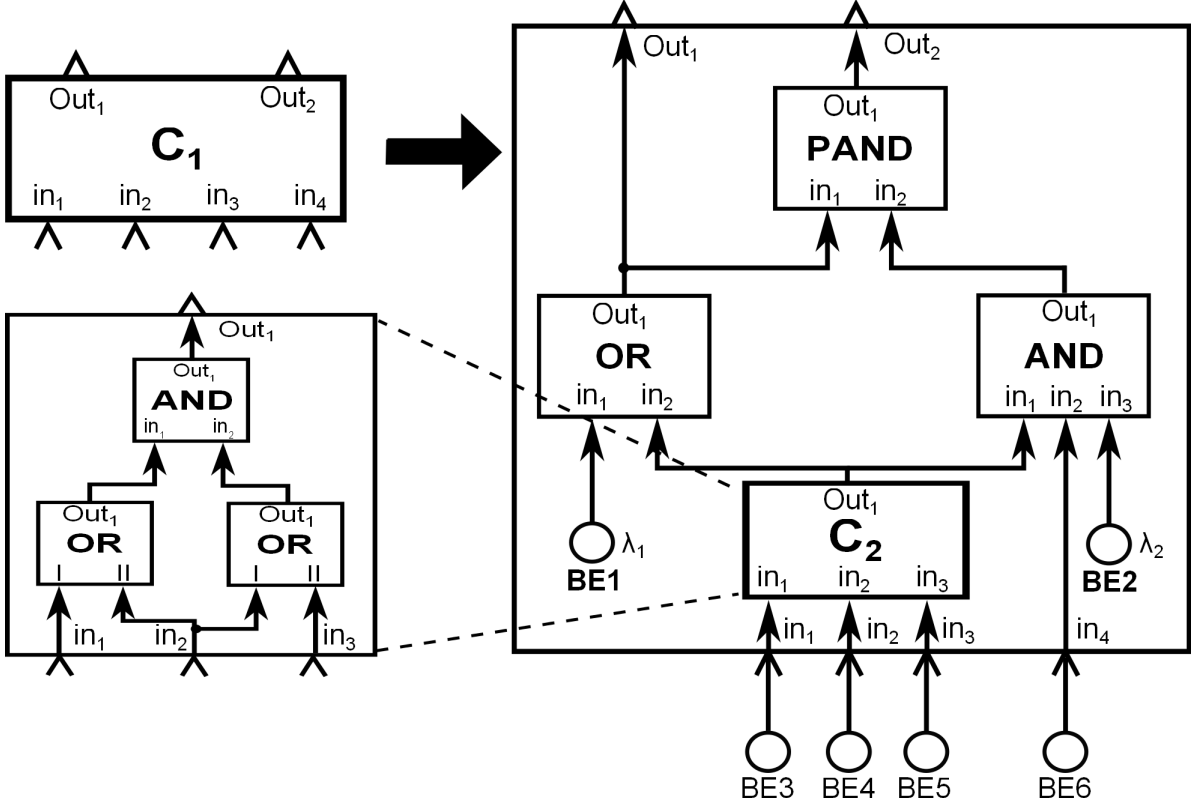


Figure 4.2: Component Dynamic Fault Tree Overview

Definition 4.1. Component Dynamic Fault Tree: the component dynamic fault tree model, $cdft$, is a 4-tuple $\langle N, G, SC, E \rangle$ where:

- N is the set of Nodes, which are partitioned into a set of: internal events N_{intern} , input ports N_{in} and output ports N_{out} ; $N = \{N_{intern}, N_{in}, N_{out}\}$. For instance, for the CDFT model depicted in Figure 4.2, considering C_1 : $N_{intern} = \{C_1.BE1, C_1.BE2\}$, $N_{in} = \{C_1.in_1, C_1.in_2, C_1.in_3, C_1.in_4\}$, $N_{out} = \{C_1.Out_1, C_1.Out_2\}$.
- G is the set of Gates, where each gate $g \in G$ is described by: one output port $g.out$; one or more input ports $g.in_i / i \in \mathbb{N}$; a dynamic function which links inputs with outputs according to static (AND, OR, KooN) and/or dynamic (PAND) Fault Tree gates. As displayed in Table 4.3, the behaviour of the CDFT gates are characterized according to its input events (A,B) , which can be extended to an arbitrary number

of input events.

- A set SC of Sub-Components, where each subcomponent $sc \in SC$ is described by: one or more output ports $sc.out_i$; one or more input ports $sc.in_i$; and a mapping to another CDFT component's failure logic. For instance, for the CDFT model depicted in Figure 4.2, $SC=C_2$: $N_{in} = \{C_2.in_1, C_2.in_2, C_2.in_3\}$, $N_{out} = \{C_2.Out_1\}$; mapping: $C_1.in_1 \rightarrow C_2.in_1$; $C_1.in_2 \rightarrow C_2.in_2$; $C_1.in_3 \rightarrow C_2.in_3$; $C_2.out_1 \rightarrow OR.in_2$; $C_2.out_1 \rightarrow AND.in_1$;
- A set of directed Edges $E \subseteq ((N_{intern} \cup N_{in} \cup G.OUT \cup SC.OUT) \times (N_{out} \cup G.IN \cup SC.IN))$ where: $G.OUT$ is set of all outputs of all gates; $G.IN$ is set of all inputs of all gates; $SC.OUT$ is the set of all outputs of all sub-components; and $SC.IN$ is the set of all inputs of all sub-components.

Table 4.3: Component Dynamic Fault Tree Gates

Gate Notation	(Gate Behaviour)
$Y=AND(A,B)$	If A fails and B fails, then Y fails
$Y=OR(A,B)$	If A fails or B fails, then Y fails
$Y=PAND(A,B)$	If A fails before the failure of B or at the same time, then Y fails
$Y=NOT(A)$	If A doesn't fail, then Y fails

CDFT components describe the failure logic of a component through temporal and/or boolean functions, determining the occurrence of the output events depending on the input event occurrences and its corresponding occurrence time.

While a basic event characterizes self-contained simple failure logic, a component encloses any-complexity failure logic (with possibly multiple I/O dependencies) specified using BEs, gates, and further sub-components. Therefore, the CDFT paradigm makes it possible to embed in a component the *dynamic* failure logic of a (sub)system and (re)use it where needed addressing repeated components and repeated basic events.

Figure 4.2 characterizes a hypothetical CDFT model with repeated components (C_2) and CDFT gates. Each component (C_1, C_2) may have gates, basic events and/or other components as inputs. Each basic event (BE_1, BE_2, \dots, BE_6) is characterized according to its probability density function and its failure rates. The failure rates may be specified

as a single value or interval of possible failure rate values allowing to understand their influence on system failure behaviour (see Section 4.4).

The failure evaluation algorithm for the model in Figure 4.2 is:

$$\begin{aligned}
 C_2.Out_1 &= \mathbf{AND}(\mathbf{OR}(BE3, BE4), \mathbf{OR}(BE4, BE5)) \\
 C_1.Out_1 &= \mathbf{OR}(\mathbf{BE}(\lambda_1, 'exponential'), C_2) \\
 C_1.Out_2 &= \mathbf{PAND}(\mathbf{OR}(\mathbf{BE}(\lambda_1, 'exponential'), C_2), \mathbf{AND}(C_2, BE6, \mathbf{BE}(\lambda_2, 'exponential')))
 \end{aligned}$$

where the function $\mathbf{BE}(\text{parameters}, \text{distribution})$ generates the corresponding failure data of basic events. Note: $C_2.Out_1$ is simplified to C_2 in the previous equations because C_2 has a single output.

This approach (as with the Component Fault Tree approach) enables the system refinement through architectural components until reaching an indivisible component, instead of the classical top-down approach adopted in most of the Fault Tree implementations.

To *implement* the CDFT paradigm, Monte Carlo simulations are performed on the system's failure evaluation algorithm in order to estimate the failure probability. To this end, it is executed a large number of times, each execution comprising of a set of random variables corresponding to the failure occurrences of the basic events. From the law of big numbers, in the long run the failure probabilities of the system are calculated throughout its lifetime [Zio13]. For each execution: (1) the random time to failure of basic events are calculated according to their cumulative probability distribution function; (2) connected gates and/or components use this information to determine their outcome (functional or failed state); (3) When a failure at the output of a gate or component occurs, the failure time information is passed to the next gate/component so that the system's dynamic failure logic is tracked from basic events to system-level top-event.

In the Component Fault Tree approach it is possible to reuse a component throughout the model. With Component Dynamic Fault Trees the same concept is applied through the reuse of the outcomes of system gates/subcomponents and their inner CDFT failure logic and basic events. While for the implementation of CFTs combinatorial logic and algorithms for the evaluation of binary decision diagrams are applied (see Subsection

2.3.1), in order to solve CDFT models Monte Carlo simulations are used accounting for the temporal occurrence of events and components.

Figure 4.3 depicts an example model that shows how the CDFT is implemented using repeated components (IE4, IE5) and repeated events (BE2, BE5). The CDFT model improves the readability and manageability of the dynamic model. See Appendix C to see how to model the same example using other formalisms.

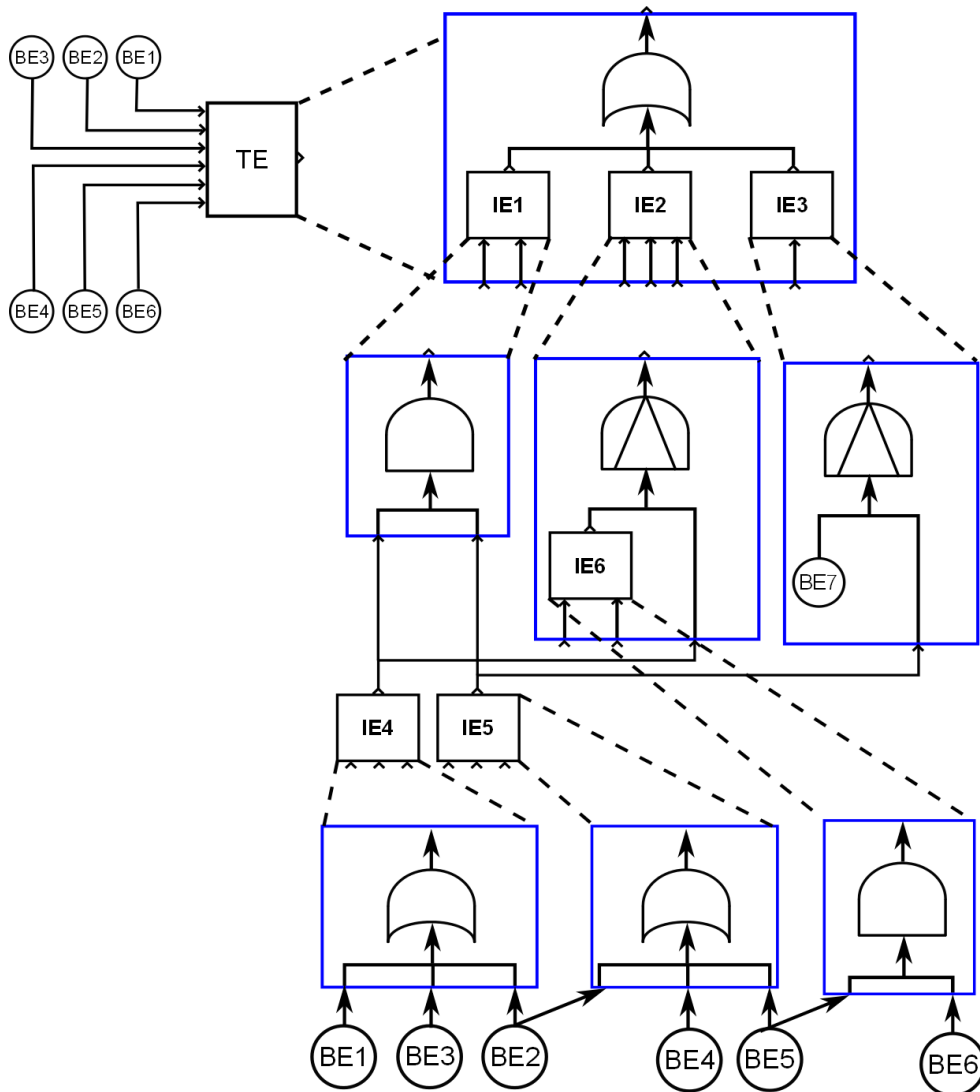


Figure 4.3: Component Dynamic Fault Tree Example

To analyse CDFTs, MatCarloRe tool [Manno12b] has been extended with NOT gates, importance measurements (cf. Section 4.3) and uncertainty analyses (cf. Section 4.4).

4.3 Sensitivity Analysis

The goal of the sensitivity analysis (importance measurement) is to weigh the contribution of components (or basic events) to the top-event failure occurrence based on the structure of a system and component reliability. In the next subsections we explain simulation-based methods to estimate importance measurement indices and then how we implemented the chosen method.

There exist analytical importance measurement indices [vanderBorst01] and they have been applied successfully by [Fricks03] and [Ou00]. However, since Component Dynamic Fault Trees are analysed through Monte Carlo simulations, in this dissertation only simulation-based importance measurement index values have been considered.

4.3.1 Simulation-based Importance Measurement Indices

Owing to the increasing complexity of current systems, in some cases analytical calculations of important measurements are not feasible. To overcome this issue, simulation-based importance measurements were introduced [Wang04].

Failure Criticality Index (FCI): FCI value indicates the contribution (percentage) of the i -th component's failure to the system overall failure:

$$\mathcal{I}_i^{FCI} = \frac{n_i^F}{N^F} \quad (4.15)$$

where,

- \mathcal{I}_i^{FCI} : failure criticality index of the i -th component.
- n_i^F : number of system failures caused by component i .
- N^F : total number of system failures.

To evaluate the frequency of the i -th component failure causing a system failure, we record the number of system failures caused by the component i (n_i^F) with respect to the total number of system failures (N^F).

Other simulation-based measurements focus on repair characteristics and component's uptime/downtime values:

Restore Criticality Index (RCI): RCI value indicates the contribution (percentage) of the i -th component's repair to the system's overall repair:

$$\mathcal{I}_i^{RCI} = \frac{n_i^R}{N^R} \quad (4.16)$$

where,

- \mathcal{I}_i^{RCI} : restore criticality index of the component i .
- n_i^R : number of system repairs caused by component i .
- N^R : total number of system repairs.

Operational Criticality Index (OCI): OCI value is defined as the percentage of the i -th component's downtime over the system downtime:

$$\mathcal{I}_i^{OCI} = \frac{down_i}{down_{sys}} \quad (4.17)$$

where,

- \mathcal{I}_i^{OCI} : operational criticality index of the i -th component.
- $down_i$: downtime of the component i .
- $down_{sys}$: system downtime.

Due to the architectural design assumptions (non-repairable resources) importance measurements which consider repair characteristics are not considered (restore criticality index). The operational criticality index measures downtime values of the system and of a component, but it does not consider the contribution of the component to the system failure. Therefore, we focus on the Failure Criticality Index (\mathcal{I}_i^{FCI}) due to its direct application with CDFTs and the significance of its measurements. The main goal of the FCI measurement is to identify weaknesses of the system design.

4.3.2 Implementation of the Sensitivity Analysis

To implement the failure criticality index evaluation, we resort to the gates of the CDFT model. For each input event (E_1, E_2, \dots, E_N) , their failure criticality index values are calculated by examining when the input event E_i has caused the occurrence of a gate's output event (Y). Subsequently, this analysis is extended to the output of the next gate until we reach the output of the component and this process is repeated until the output of the system. In this way, we obtain the chain of gates and components that cause the top-event of the system. For each Monte Carlo trial, the components causing the top-event's failure occurrence are recorded and after a total of N Monte Carlo trials, the relation between: (1) the total number of times the output event occurs due to the failure of an input event and (2) the total number of output event failure occurrences is calculated.

There exist two alternatives for considering the system failures caused by event/component i :

- Last event that caused the system to fail - triggering event.
- Minimal cut-set.

In our implementation, we have considered the triggering event implementation. As noted by [Hilber05], the rationale under this decision relies on the fact that the index becomes non-ambiguous and it is not necessary to calculate minimal cut-sets.

It is assumed that an input event causes the occurrence of the output event when the input event's occurrence time (uptime) matches with the output event's occurrence time (uptime) (cf. Figure 4.4). With the OR gate logic, it is necessary to take into account top event's downtime: if the event that caused the top event occurrence (uptime) is no longer failed but the top event continues to be failed top event's downtime needs to be checked (cf. Figure 4.4 dashed line event).

Therefore, when analysing complex systems, the system CDFT will have a set of interconnected gates and each of them will have its own failure criticality index values with respect to its input events.

Algorithm 1 determines for a CDFT model the failure criticality index of any of its

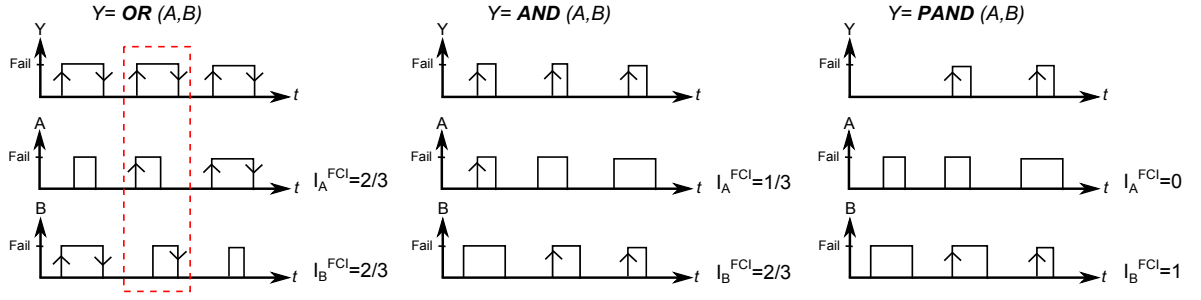


Figure 4.4: FCI Example Time-Diagrams

constituent components or events. To this end the algorithm requires:

- The *event* whose failure criticality index needs to be evaluated (represented by its name, e.g., *BE2 failed*).
- The *Top Event (TE)* gate whose failure criticality index needs to be evaluated. The gate is represented by the following information: the output event's name and a list of input events that are used to evaluate the *TE* output.
- A vector with the information for all the gates of the CDFT model. Each gate has its output event's name and the list of input events that are used to evaluate its outputs.

The output name of a *gate* will be referenced by *gate.Output_Name* in the algorithm. The list of input events will be referenced by *gate.Input(j)*, where *j* goes from 1 to the number of inputs of the *gate*. Finally, each *gate.Input(j)* contains its name (*gate.Input(j).Input_Name*), and the FCI of this input event with respect to the output event of this gate (*gate.Input(j).FCI_Value*), i.e., the percentage that this input was the cause of the output failure.

The Algorithm 1 traverses the CDFT's structure in a top-down manner finding the failure criticality index value of the *event* variable compositionally: the contribution of each intermediate event (or a basic event) is weighted according to the contribution of the gate they belong to (see Figure 4.5 and its explanations).

The system example in Figure 4.5 shows the application of the Algorithm 1 to the hypothetical system shown in Figure 4.3 and in Appendix C.

System's failure event *TE* is caused by %IE1, %IE2 and %IE3; accordingly, each of them

Algorithm 1 Criticality Analysis

```
1: function system_fci = FCI(event, TE, subtree)
2:   system_fci = 0;
3:   i = 0;
4:   done = 0;
5:   // for each branch of the tree starting from the TE
6:   while (i <= length(TE.Input)) AND (!done) do
7:     fci = 1; // init for each branch
8:     IE = TE.Input(i);
9:     fci = fci * IE.FCI_Value; // FCI of the corresponding input
10:    if strcmp(IE.Input_Name, event) then // is this the analysed event?
11:      done = 1;
12:    else // not matching with the 1st level, try inner subtree
13:      j = 0;
14:      inner = 0;
15:      while (!inner) AND (j < length(subtree)) do
16:        j = j + 1;
17:        if strcmp(subtree(j).Output_Name, IE.Input_Name) then
18:          inner = 1; // there is an inner event
19:        if (inner) then
20:          branch = subtree(j); // new TE to be found
21:          branch_fci = FCI(event, branch, subtree(:)); // recursive call
22:          fci = fci * branch_fci; // update branch FCI contribution to TE
23:        if (inner) OR (done) then
24:          system_fci = system_fci + fci; // sum branch to system FCI
25:      i = i + 1;
return system_fci
```

will be caused also by its underlying intermediate events (e.g., %IE1 is caused by %IE4 and %IE5) until reaching the basic event level (e.g., %IE4 is caused by %BE1, %BE2 and %BE3). Therefore, the failure criticality index for the BE2 is calculated as follows:

$$\mathcal{I}_{BE2}^{FCI} = (\%IE1).(\%IE4).(\%BE2) + (\%IE1).(\%IE5).(\%BE2) + (\%IE2).(\%IE7).(\%BE2) + (\%IE3).(\%IE8).(\%BE2)$$

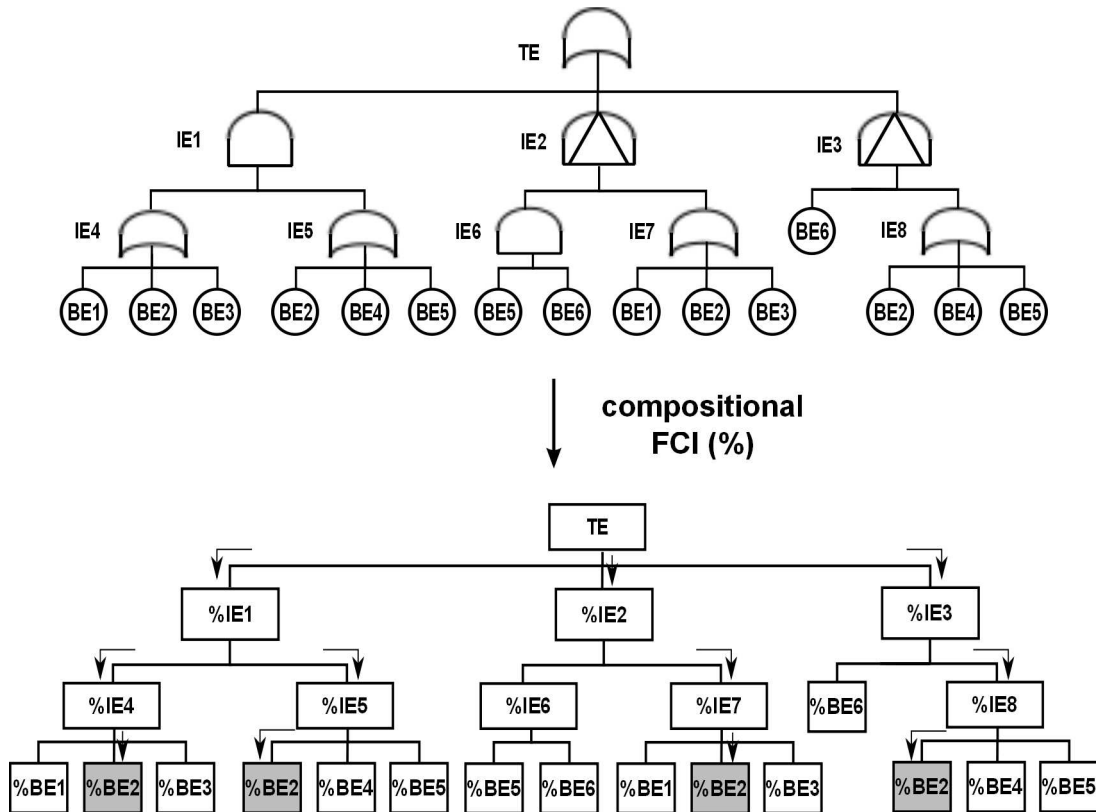


Figure 4.5: Failure Criticality Index Calculation Example

4.4 Uncertainty Analysis

The inability to obtain statistical failure characteristics of certain components hampers the applicability of the Dependability Evaluation Modelling approach. In order to deal with uncertain failure data of components, uncertainty analyses have been integrated within the D3H2 methodology.

It has been demonstrated that software failure rates are difficult to determine (e.g., see [Littlewood00b; Goševa-Popstojanova01; Lyu07]). This uncertainty in parameter estimation can lead to very different dependability analysis results.

There exist different sources of uncertainty [Oberkampff04]:

- Epistemic Uncertainty: lack of knowledge or information in any phase or activity of the modelling process.

- Aleatory Uncertainty: inherent variation associated with the physical system or environment under consideration.

In this dissertation we are concerned with the epistemic uncertainty, which deals with the lack of knowledge of the exact behaviour of the system. The epistemic uncertainty will be addressed/considered as the influence of uncertain component parameter values (failure rate) on system's unreliability value.

In order to deal with the lack of exact knowledge of the failure rate data, second-order probabilities (i.e., statistic distribution of failure occurrence probabilities) have been implemented in the MatCarloRE tool [Manno12b], allowing to calculate second-order probability mass functions of: system failure probabilities and importance (sensitivity) measurements.

Each basic event is modelled with its corresponding random variable according to its failure distribution and parameters. When integrating uncertainty in the DEM the implemented approach allows the designer to specify interval failure rates: (1) the random number corresponding to every variables' failure rate interval is sampled randomly, (2) then the corresponding probability of interest is calculated, and (3) finally outcome probabilities are distributed among histogram bins (where relative number of samples per bin indicates the probability of the bin's associated probability interval) resulting in a probabilistic distribution of probability values [Forster09] - second order probabilities (cf. Figure 4.6). For simplicity and due to the lack of knowledge of real failure data values, the stochastic distribution of variable probability intervals is chosen to be uniform.

The following main activities are involved in the uncertainty analysis process:

1. Monte Carlo sampling of the uncertain variables: from the failure rates of the uncertain variables - specified as interval values - a single failure rate value is chosen randomly within the specified failure rate interval according to the uniform distribution. The outcome of this activity is a randomly sampled failure rate.
2. Monte Carlo sampling of the time to failure (occurrence) of both uncertain variables and known variables based on their failure rate values. The outcome of this process are a set of randomly sampled time to failures.
3. With the updated numerical values of data variables, the Component Dynamic

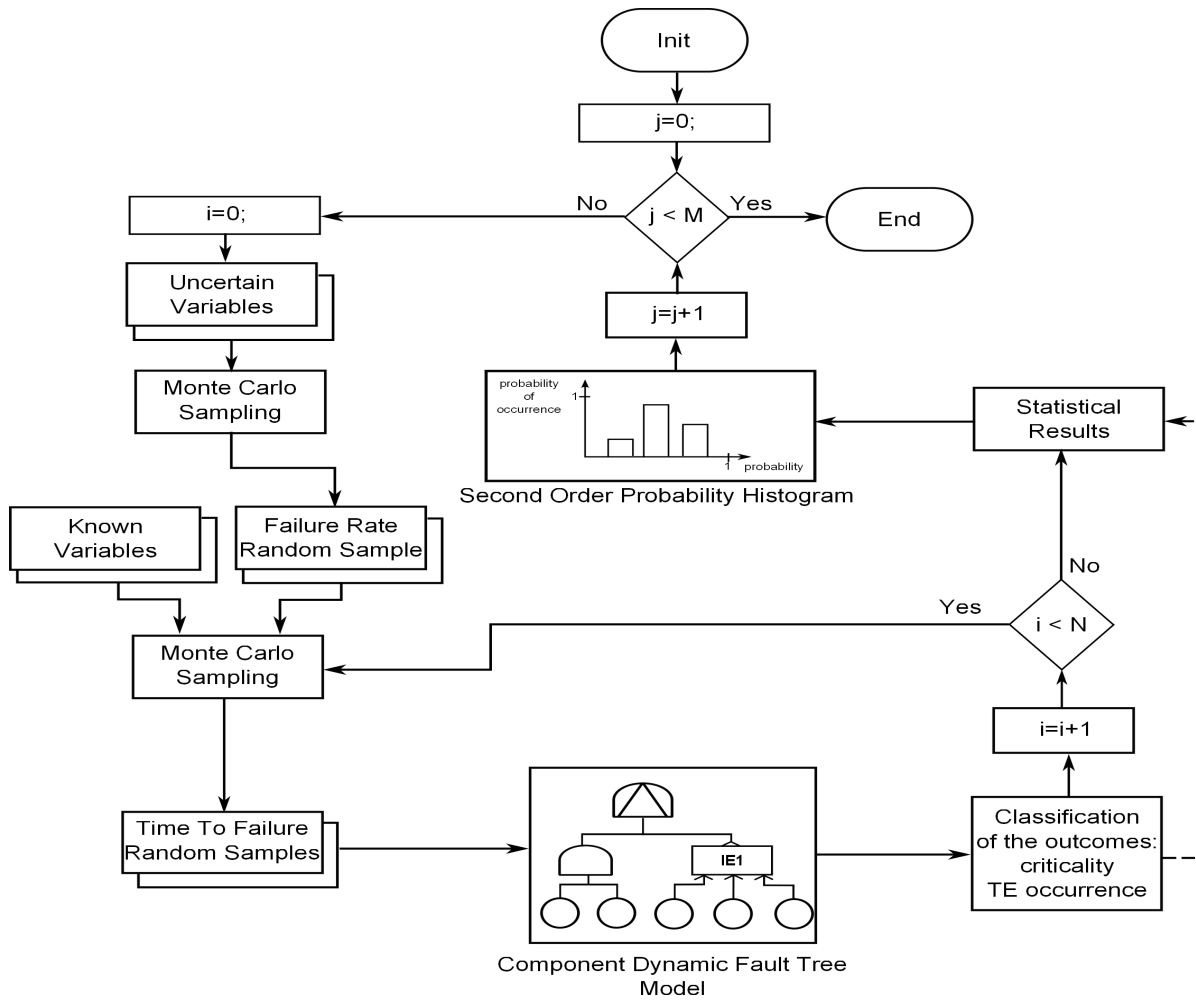


Figure 4.6: Overview of the Uncertainty Analysis

Fault Tree model is solved getting as an outcome counters of top-event failure occurrences and critical event failure occurrences gathered in a system's mission time (lifetime) vector.

4. If the threshold of N Monte Carlo trials is reached, accumulated Component Dynamic Fault Tree model's statistical results are gathered in a histogram which counts and classifies the (probabilistic) frequency of occurrence of the top event.
5. If the threshold of M Monte Carlo trials is reached, the overall process ends up and the histogram is normalized. Otherwise, the process is restarted again by sampling failure rate values randomly and time to failure of the basic events.

The main drawback of this approach is the time needed for the computation of Monte

Carlo simulations ($M \times N$ iterations in Figure 4.6). While there exist analytic techniques for the reduction of this time (e.g., dynamic stopping criterion [Meedeniya11]), we have opted for using Matlab's parallel toolbox in order to perform parallel tasks in several computers at a time.

4.5 Cost Analysis

The cost assessment is carried out by adding up the cost of hardware and software resources (see Appendix E for the specific values).

Software costs: the cost of software components is quantified by considering their development cost assuming that it will be paid off in X^6 years. We classify 4 types of software components: fault detection software (SW_FD), reconfiguration software (SW_R), reconfiguration's fault detection software (SW_FD_R) and Control/Detector software (SW_Det).

The development costs for each of these 4 SW components is considered once for different subfunction implementations of the same main function: once developed, they are adapted for the related SF implementations. This assumption is adopted because the grouped subfunction implementations are closely related and they do not need a significant development cost (as demonstrated in [Kanoun01] through an empirical case study, the cost of N variants (in design diversity) is not N times the cost of a single software variant): (1) fault detection implementations adapt to different subfunctions modifying subfunction-specific time/value thresholds; (2) reconfiguration implementations' development cost does not differ for different subfunctions, alternative implementations will have allocated different reconfiguration tables for different subfunctions, but reactivation logic holds the same for different subfunction's reconfiguration implementations; (3) reconfiguration's fault detection implementations development cost for different subfunctions differ only in the keepalive timeout, but their development is independent of any subfunction; and (4) all the considered control/detector software implementations have a closely related logic.

Hardware cost: the cost estimation of sensors, controllers and actuators can be ob-

⁶Let us assume $X=4$ years for calculation purposes.

tained from their suppliers. Human cost related with mounting and testing tasks is considered for sensors and actuators assuming 10 minutes per sensor (actuator) at a rate of 60 €/hour.

4.6 Results

Taking the *extended HW/SW architectures* of the safety-critical Door Status Control and Fire Protection Control main functions as a starting point (see Table 3.10 and Table 3.14 for the *extended HW/SW architectures* of the Fire Protection Control and the Door Status Control main functions respectively), the Dependability Evaluation Algorithm is applied to both main functions. Furthermore, criticality analysis to evaluate the robustness of different redundancies and uncertainty analysis to manage the lack of failure data information of software and communication resources are implemented as well. Resultantly, in Subsection 4.6.1 and Subsection 4.6.2 the Dependability Evaluation Models for the Fire Protection Control and Door Status Control main functions are presented respectively. The failure rates and cost values of the different resources are presented in Appendix E.

4.6.1 Fire Protection Control

In this subsection different design strategies are analysed with respect to dependability and cost for the Fire Protection Control main function [Aizpurua14]. By means of the dependability evaluation model, simulations are performed to evaluate: (1) redundancy strategies; (2) reconfiguration strategies; and (3) validity of the hypothesis of the ideal behaviour of fault detection, reconfiguration and communication.

Dependability Evaluation Model

According to the DEM approach, subfunction's implementations are characterized with the failure rates of its constituent resources. For the Fire Detection subfunction (cf. Table 3.10, implementations #2 and #3), its implementation failures are specified as

follows⁷ (see Equation 4.1 for more information about the λ notation):

$$\begin{aligned}\mathcal{F}_{\text{FireDetection}_1} &= \mathbf{OR}(\lambda_{\text{PU}_{\text{FP}}}, \lambda_{\text{FireDetector}}) \\ \mathcal{F}_{\text{FireDetection}_2} &= \mathbf{OR}(\lambda_{\text{PU}_{\text{ACC_A}}}, \lambda_{\text{TemperatureSensor}}, \lambda_{\text{SW}_{\text{FireDetection}}}, \lambda_{\text{Comm}})\end{aligned}$$

The same equation holds for the failure characterizations of the omission failures of: fault detection of the fire detection ($\mathcal{F}_{\text{FD_FireDetection}_1 \text{ O - \#4}}$), reconfiguration of the fire detection ($\mathcal{F}_{\text{R_FireDetection}_1 \text{ O - \#5}}$, $\mathcal{F}_{\text{R_FireDetection}_2 \text{ O - \#6}}$), and fault detection of the fire detection's reconfiguration ($\mathcal{F}_{\text{FD_R_FireDetection}_1 \text{ O - \#7}}$, $\mathcal{F}_{\text{FD_R_FireDetection}_2 \text{ O - \#8}}$) subfunctions implementations:

$$\begin{aligned}\mathcal{F}_{\text{FD_FireDetection}_1 \text{ O}} &= \mathbf{OR}(\lambda_{\text{PU}_{\text{FP}}}, \lambda_{\text{SW}_{\text{FD_FireDetection}}}, \lambda_{\text{Comm}}) \\ \mathcal{F}_{\text{R_FireDetection}_1 \text{ O}} &= \mathbf{OR}(\lambda_{\text{PU}_{\text{FP}}}, \lambda_{\text{SW}_{\text{R_FireDetection}}}) \\ \mathcal{F}_{\text{R_FireDetection}_2 \text{ O}} &= \mathbf{OR}(\lambda_{\text{PU}_{\text{ACC_A}}}, \lambda_{\text{SW}_{\text{R_FireDetection}}}, \lambda_{\text{Comm}}) \\ \mathcal{F}_{\text{FD_R_FireDetection}_1 \text{ O}} &= \mathbf{OR}(\lambda_{\text{PU}_{\text{ACC_A}}}, \lambda_{\text{SW}_{\text{FD_R_FireDetection}}}, \lambda_{\text{Comm}}) \\ \mathcal{F}_{\text{FD_R_FireDetection}_2 \text{ O}} &= \mathbf{OR}(\lambda_{\text{PU}_{\text{FP}}}, \lambda_{\text{SW}_{\text{FD_R_FireDetection}}}, \lambda_{\text{Comm}})\end{aligned}$$

Accordingly, the false positive failures will be characterized with their characterizing failure distribution and corresponding parameters: $\mathcal{F}_{\text{FD_FireDetection FP}} = \lambda_{\text{FD_FireDetection FP}}$; $\mathcal{F}_{\text{FD_R_FireDetection}_1 \text{ FP}} = \lambda_{\text{FD_R_FireDetection}_1 \text{ FP}}$; $\mathcal{F}_{\text{FD_R_FireDetection}_2 \text{ FP}} = \lambda_{\text{FD_R_FireDetection}_2 \text{ FP}}$.

The failure of the fire detection subfunction will be characterized according to the following equation:

$$\mathcal{F}_{\text{FireDetection}} = \mathbf{OR}(\mathcal{F}_{\text{All Impl_FireDetection}}, \mathcal{F}_{\text{Unresolved_FireDetection}}, \mathcal{F}_{\text{Dependencies_FireDetection}})$$

The $\mathcal{F}_{\text{All Impl_FireDetection}}$ event will happen when each implementation fails or is detected as failed:

$$\mathcal{F}_{\text{All Impl_FireDetection}} = \mathbf{AND}(\mathcal{F}_{\text{FireDetection}_1 \text{ FP}}, \mathcal{F}_{\text{FireDetection}_2 \text{ FP}})$$

where $\mathcal{F}_{\text{FireDetection}_i \text{ FP}} = \mathbf{OR}(\mathcal{F}_{\text{FireDetection}_i}, \lambda_{\text{FD_FireDetection FP}})$; $i = \{1, 2\}$.

⁷For the sake of simplification we will include in λ_{Comm} failure rates of all the communication networks and interconnecting gateway device.

Since the fire detection subfunction has 2 implementations, the failure unresolved event will take into account the failure unresolved situation of the first implementation:

$$\mathcal{F}_{\text{Unr. Impl}_1\text{FireDetection}} = \mathbf{OR}(\mathcal{F}_{\text{R Seq.}_1\text{FireDetection}}, \mathcal{F}_{\text{FD Seq.}_1\text{FireDetection}})$$

The reconfiguration sequence failure and fault detection sequence failure for the first implementation of the fire detection subfunction are defined as follows:

$$\mathcal{F}_{\text{R Seq.}_1\text{FireDetection}} = \mathbf{PAND}(\mathcal{F}_{\text{R FireDetection}}, \mathcal{F}_{\text{FireDetection}_1 \text{FP}})$$

The reconfiguration failure $\mathcal{F}_{\text{R FireDetection}}$ is developed as follows:

$$\mathcal{F}_{\text{R FireDetection}} = \mathbf{OR}(\mathcal{F}_{\text{All R Impl. FireDetection}}, \mathcal{F}_{\text{R Unresolved FireDetection}})$$

where,

$$\mathcal{F}_{\text{All R Impl. FireDetection}} = \mathbf{AND}(\mathcal{F}_{\text{R FireDetection}_1 \text{O/FP}}, \mathcal{F}_{\text{R FireDetection}_2 \text{O/FP}})$$

$$\mathcal{F}_{\text{R FireDetection}_i \text{O/FP}} = \mathbf{OR}(\mathcal{F}_{\text{R FireDetection}_i \text{O}}, \lambda_{\text{R FireDetection}_i \text{FP}}); \quad i=\{1,2\}$$

$$\mathcal{F}_{\text{R Unresolved FireDetection}} = \mathbf{AND}(\mathcal{F}_{\text{FD R}_1 \text{O}}, \mathcal{F}_{\text{FD R}_2 \text{O}})$$

The fault detection sequence failure for the fire detection subfunction is defined as follows:

$$\mathcal{F}_{\text{FD Seq. FireDetection}_1} = \mathbf{PAND}(\mathcal{F}_{\text{FD FireDetection}}, \mathcal{F}_{\text{FireDetection}_1})$$

The fault detection failure of the fire detection $\mathcal{F}_{\text{FD FireDetection}}$ depends on the operation of the destination subfunction (SF_{DEST}), because the FD implementation is located at the same PU:

$$\mathcal{F}_{\text{FD FireDetection}} = \mathcal{F}_{\text{FD Dest}_1}$$

The destination subfunction is the Fire Control Algorithm (FCA) subfunction (implementation #9 in Table 3.10):

$$\mathcal{F}_{\text{FD_Dest}_1} = \mathbf{OR}(\mathcal{F}_{\text{FireControlAlgorithm}_1}, \mathcal{F}_{\text{FD_FireDetection}_1 \text{ O}})$$

where,

$$\mathcal{F}_{\text{FireControlAlgorithm}_1} = \mathbf{OR}(\lambda_{\text{PU}_{\text{FP}}}, \lambda_{\text{SW}_{\text{FireControl}}}, \lambda_{\text{Comm}})$$

Note that we have avoided including fire control algorithm subfunction's dependencies at this level (i.e., user emergency signal and fire detection subfunctions) because it would create a logical loop. Dependencies are taken into account at a higher level (see fire control algorithm subfunction failure's characterization - $\mathcal{F}_{\text{FireControlAlgorithm}}$).

There is no input dependency for the fire detection subfunction ($\mathcal{F}_{\text{Dependencies_FireDet}} = 0$): it is an input subfunction and therefore, it does not receive data from another subfunction.

The user emergency signal input subfunction (cf. Table 3.10 #1) does not have redundancies. Therefore, its failure characterization is directly obtained through the failure characterization of the implementation's constituent resources:

$$\mathcal{F}_{\text{UserEmergencySignal}} = \mathcal{F}_{\text{UserEmergencySignal}_1} = \mathbf{OR}(\lambda_{\text{Emergency Button}}, \text{PU}_{\text{FP}})$$

As for the fire control algorithm, there are no implementation redundancies, but there exist input dependencies. Therefore, its failure expression is as follows:

$$\mathcal{F}_{\text{FireControlAlgorithm}} = \mathbf{OR}(\mathcal{F}_{\text{All Impl_FireControlAlgorithm}}, \mathcal{F}_{\text{Dependencies_FireControlAlgorithm}})$$

where,

$$\begin{aligned} \mathcal{F}_{\text{All Impl_FireControlAlgorithm}} &= \mathcal{F}_{\text{FireControlAlgorithm}_1} \\ \mathcal{F}_{\text{Dependencies_FireControlAlgorithm}} &= \mathcal{F}_{\text{Dep. C_CL}} \\ \mathcal{F}_{\text{Dep. C_CL}} &= \mathbf{AND}(\mathcal{W}_{\text{C_CL}}, \mathcal{F}_{\text{I_CL}}) \\ \mathcal{W}_{\text{C_CL}} &= \mathbf{NOT}(\mathcal{F}_{\text{FireControlAlgorithm}_1}) \\ \mathcal{F}_{\text{I_CL}} &= \mathbf{OR}(\mathcal{F}_{\text{UserEmergencySignal}}, \mathcal{F}_{\text{FireDetection}}) \end{aligned}$$

Therefore, after simplification⁸, the fire control algorithm subfunction's failure is specified as follows:

$$\mathcal{F}_{\text{FireControlAlgorithm}} = \mathbf{OR}(\mathcal{F}_{\text{FireControlAlgorithm}_1}, \mathcal{F}_{\text{UserEmergencySignal}}, \mathcal{F}_{\text{FireDetection}})$$

Finally, the failure of the fire extinction subfunction ($\mathcal{F}_{\text{FireExtinction}}$) and accordingly, the failure of the Fire Protection Control main function is specified as follows:

$$\mathcal{F}_{\text{FireExtinction}} = \mathbf{OR}(\mathcal{F}_{\text{All Impl. FireExtinction}}, \mathcal{F}_{\text{Unresolved FireExtinction}}, \mathcal{F}_{\text{Dependencies FireExtinction}})$$

Note that the fire extinction subfunction has one implementation (#10), therefore: $\mathcal{F}_{\text{All Impl. FireExtinction}} = \mathcal{F}_{\text{FireExtinction}_1}$ and $\mathcal{F}_{\text{Unresolved FireExtinction}} = 0$.

$$\begin{aligned} \mathcal{F}_{\text{FireExtinction}} &= \mathbf{OR}(\mathcal{F}_{\text{FireExtinction}_1}, \mathcal{F}_{\text{FireControlAlgorithm}}) \\ \mathcal{F}_{\text{FireExtinction}_1} &= \mathbf{OR}(\lambda_{\text{PU}_{\text{FP}}}, \lambda_{\text{Sprinkler}}) \end{aligned}$$

Redundancy Strategies

To evaluate the failure probability of the Fire Protection Control main function's architecture combinations, the architecture configurations displayed in Table 4.4 have been tested.

Table 4.4: Fire Protection Control Configurations with Alternative Redundancy Strategies

ID	Configuration
#1	No redundancies (cf. Table 3.8)
#2	1 Heterogeneous redundancy (cf. Table 3.10)
#3	1 Homogeneous redundancy connected to the same PU_{FP}
#4	1 Homogeneous redundancy connected to a different PU

Figure 4.7 depicts Fire Protection Control configurations' relative failure probability

⁸ $A + \overline{A}.B = A + B$

and relative cost normalized with respect to the configuration without redundancies. Alternative *extended HW/SW architectures* are analysed adding a homogeneous or heterogeneous redundancy to the fire detection subfunction. With homogeneous redundancies, the fire detection sensor has been replicated with two alternative configurations: connect both fire detection sensors to the PU_{FP} (#3) or connect each fire detection sensor to a different PU (#4). All these configurations include the same fault detection and reconfiguration implementations (cf. Table 3.10).

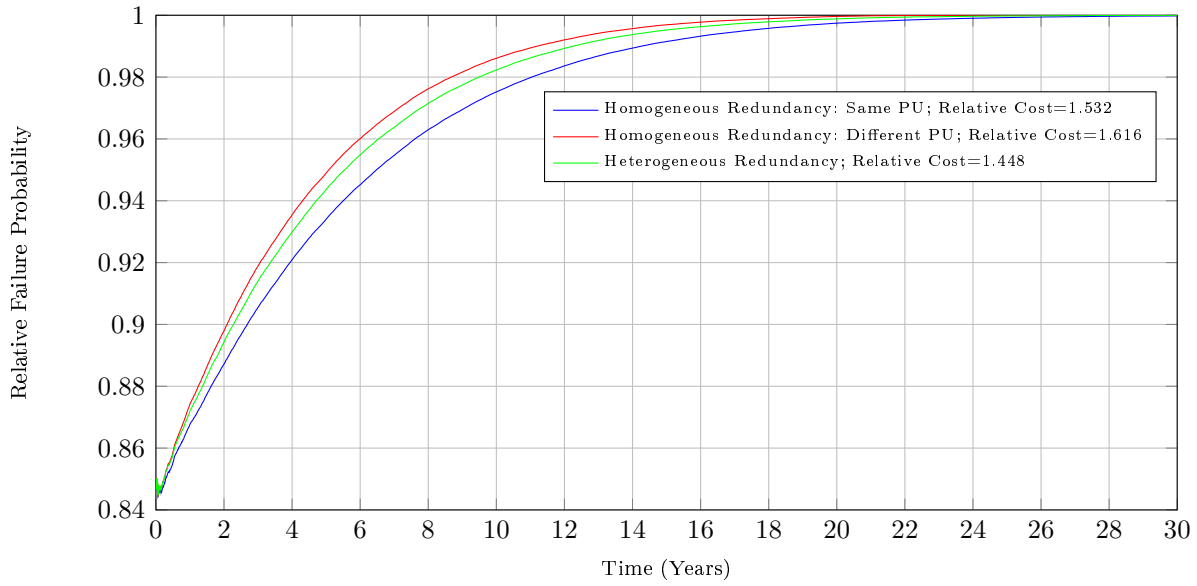


Figure 4.7: Relative Failure Probability & Cost of Fire Protection Control Configurations (10^6 iterations)

As Figure 4.7 depicts, heterogeneous redundancy configuration's failure probability is higher than homogeneous redundancy configuration when the extra homogeneous redundancy sensor is implemented in the same PU. However, when the homogeneous redundancy sensor is implemented in a additional PU, the failure probability of the architecture with homogeneous redundancy is higher. This happens because both configurations (the homogeneous redundancy implemented in a additional PU and the heterogeneous redundancy) add extra resources to the *extended HW/SW architecture*, they become more sensitive to the communication failures, and accordingly the failure probability increases. The difference between them relies on the used resources: while the heterogeneous redundancy implementation adds another PU and relies on the existing temperature sensor and the corresponding SW resource to determine the presence of fire; the homogeneous redundancy implementation adds another

PU and another fire detection sensor. The failure rate of both PUs is the same and the failure rate of the SW resources is small compared with the remainder resources ($\lambda_{SW_{FireDet}} = 1 \times 10^{-2}$), however, the failure rate of the smoke sensor ($\lambda_{FireSensor} = 3.77 \times 10^{-2}$) is higher than the temperature sensor ($\lambda_{TempSensor} = 1.49 \times 10^{-2}$).

As for the cost analysis, heterogeneous redundancies are more economical than homogeneous redundancies because this configuration reuses already existing resources in the system architecture (i.e., temperature sensor, PU).

Clearly the reliability gain is something that should be evaluated in case-by-case basis. For example, in two years time the failure probability is only reduced approximately by 10% while the cost is increased by 44% in the heterogeneous redundancies case and it is even more expensive in the other cases. In the Fire Protection Control case, the minimum number of detectors per area unit is limited by law and the addition of extra detectors would not be beneficial.

Reconfiguration Strategies

To analyse further the differences between homogeneous and heterogeneous redundancies while considering the influence of reconfiguration strategies, failure criticality index evaluations have been performed on the Fire Protection Control main function's different architectures. Namely, the reconfiguration subfunction implementations of fire detection have been duplicated: in one configuration they have been distributed in two different PUs and in another configuration they have been centralised in the same PU (cf. Table 3.10).

Table 4.5 displays the impact of the failure of redundancy and reconfiguration strategies on the system failure occurrence. The shown values are the influence of (1) fire detection subfunction's redundancy ($\mathcal{F}_{FireDetection_2}$) and (2) its reconfiguration strategies ($\mathcal{F}_{RSeq.}$) on the Fire Protection main function's failure .

Failure criticality index values provide indicators about bottleneck influences on system reliability: heterogeneous and homogeneous redundancies implemented in different PUs perform better than homogeneous redundancies located on the same PU due to the bottleneck influence on causing the top event. That is, PU_{FP} performs as a common cause failure and its failure incurs the simultaneous failure of other subfunction im-

Table 4.5: Failure Criticality Index Values of the Fire Protection Control (10^6 iterations)

Reconfiguration Strategy	Centralised			Distributed		
Redundancy Strategy	Homogeneous Same PU	Homogeneous Different PU	Heterogeneous	Homogeneous Same PU	Homogeneous Different PU	Heterogeneous
$FCL_{\mathcal{F}_{FireDetection_2}}$	0.339027	0.174606	0.179927	0.276643	0.154960	0.170669
$FCL_{\mathcal{F}_R Seq.}$	0.177554	0.171728	0.174496	0.114232	0.107315	0.106994

plementations. The same logic applies to the reconfiguration strategies: distributed reconfiguration implementations perform better than centralised implementations due to the bottleneck influence on system failure probability.

Influence of Health Management Implementations

Taking the heterogeneous redundancy configuration (#2) as a starting point (cf. Table 3.10), Figure 4.8 depicts normalized system's failure probability values (with respect to the architecture without assumptions) for different configurations under different assumptions regarding ideal fault detection, reconfiguration and communication implementations.

As Figure 4.8 shows there is a 2.5% maximum difference in relative failure probability between the real configuration and ideal configurations in which the fault detection, reconfiguration and communication are assumed to be ideal in all possible combinations. Among the ideal implementations, the configuration with the ideal communication (cf. cyan line) deviates the most from their real values. Indeed, assumptions about the ideal behaviour of the fault detection and reconfiguration implementations influence only the fire detection subfunction's performance, because fire detection is the only subfunction with redundancies within the Fire Protection Control main function. The communication influences many different subfunctions and their implementations and therefore, the assumption about the ideal behaviour of the communication plays a more important role compared with the ideal performance of the fault detection and reconfiguration implementations.

Among health management implementations, fault detection implementation has a con-

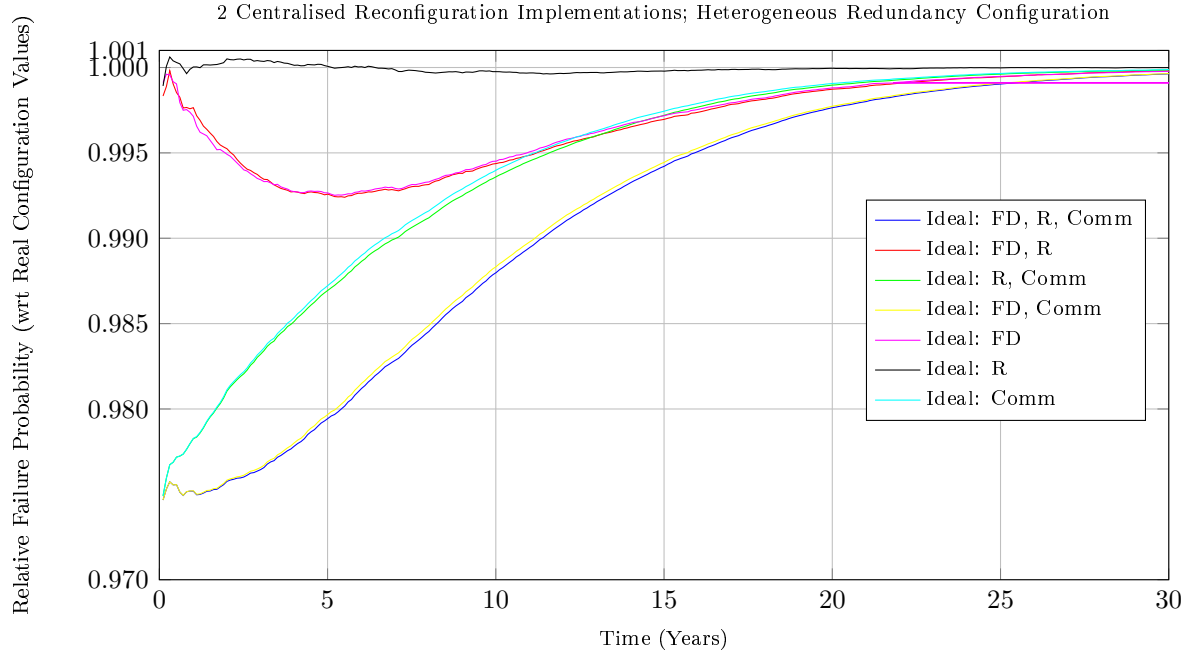


Figure 4.8: Failure Probability of Fire Protection Control Configurations under Different Assumptions (10^6 iterations)

siderable effect on the system failure probability compared with the reconfiguration implementation's influence (cf. magenta line). While the reconfiguration implementation has redundant implementations, the fault detection implementation is a single point of failure and it affects directly to the fire detection subfunction failure.

As Table 4.6 displays, we calculate the failure criticality index to (1) check the coherency of the results showed in the Figure 4.8 and (2) see the effect of the failure of different events on the Fire Protection Control main function failure. Namely, failure criticality index values of the fire control algorithm subfunction ($\mathcal{F}CI_{\mathcal{F}_{FCA}}$), fire detection subfunction ($\mathcal{F}CI_{\mathcal{F}_{FireDet}}$), fault detection sequence of the fire detection ($\mathcal{F}CI_{\mathcal{F}_{FD_FireDet\ Seq.}}$) and reconfiguration sequence of the fault detection subfunctions ($\mathcal{F}CI_{\mathcal{F}_{R_FireDet\ Seq.}}$) have been calculated. Besides, as a further reference to the previous results, the system failure probability at the time instant $T = 5$ is also displayed (Fire Extinction (FE) subfunction failure $\mathcal{F}_{FE} @ T = 5$) in the Table 4.6.

As Table 4.6 confirms, the influence of the communication's performance is considerable in conjunction with the fire detection's fault detection. Let us consider the $\mathcal{F}CI_{\mathcal{F}_{FCA}}$ column: while assuming ideal reconfiguration implementations deviates only by %0.38 from

Table 4.6: Unreliability and FCI values for Fire Protection Control Configurations under Different Assumptions (10^6 iterations)

Config.	$\mathcal{F}^{FireExtinction}$ @ T = 5	$\mathcal{FCI}_{\mathcal{F}_{FCA}}$	$\mathcal{FCI}_{\mathcal{F}_{FireDet}}$	$\mathcal{FCI}_{\mathcal{F}_{FD_FireDetSeq.}}$	$\mathcal{FCI}_{\mathcal{F}_R_FireDetSeq.}$
Ideal: FD, R, Comm	0.6497	0.3851	0.1442	0	0
Ideal: Comm, FD	0.6499	0.3869	0.2597	0	0.1725
Ideal: Comm, R	0.6547	0.3975	0.279	0.1862	0
Ideal: Comm.	0.6549	0.3995	0.2815	0.1838	0.167
Ideal: FD, R	0.6584	0.3988	0.1418	0	0
Ideal: FD	0.6585	0.4003	0.257	0	0.1725
Ideal: R	0.6634	0.4106	0.2765	0.1884	0
Real	0.6634	0.4122	0.2788	0.187	0.1744

the failure criticality index values of the real configuration, there is a %6.53 deviation if we consider the configuration associated with the ideal communication performance and ideal fault detection performance.

If we compare the failure criticality index values of the fire detection ($\mathcal{FCI}_{\mathcal{F}_{FireDet}}$) and fire protection control algorithm ($\mathcal{FCI}_{\mathcal{F}_{FCA}}$) subfunctions, we can see that the contribution of the fire detection subfunction failure to the top event's failure occurrence is reduced because: (1) other subfunctions (user emergency signal subfunction and fire protection control algorithm subfunction) also do influence to the system failure occurrence; and (2) there are repeated resources which cause the failure of different subfunctions simultaneously (e.g., PU_{FP}), contributing to their failure criticality index values altogether. This is why despite having a difference of 0.1 or greater between $\mathcal{FCI}_{\mathcal{F}_{FireDet}}$ values for different configurations, the top event's failure probability does not have considerable changes (if any).

Figure 4.9 shows the second order failure probability of the heterogeneous redundancy

configuration with 2 redundancy implementations (cf. Table 3.10) at the time instant $T = 5$ for different communication's failure rate intervals. That is, how the communication's failure rate impacts on the system's failure probability distribution.

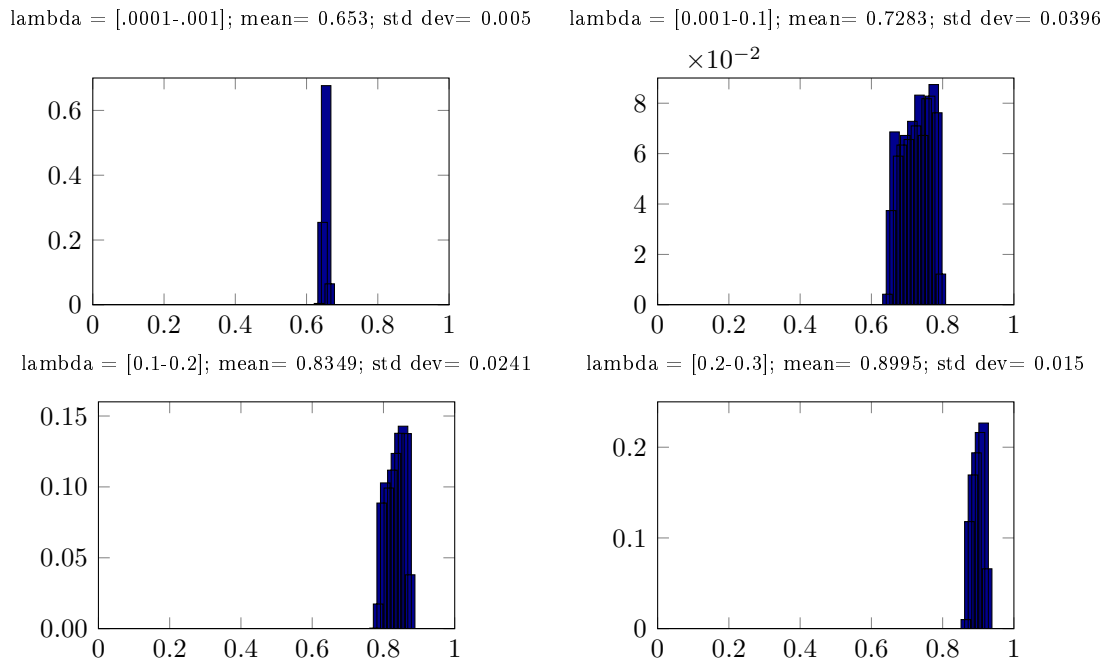


Figure 4.9: Fire Protection Control Failure Probability Distribution: Communication's Failure Rate Influence ($10^4 \times 5.10^3$ iterations)

As Figure 4.9 shows, the mean of the failure probability of the Fire Protection Control main function increases with an increment in the range of values of the communication's failure rate. These probability density function graphics show which is the impact of the communication resource's possible failure rates on the system's failure probability.

4.6.2 Door Status Control

In this subsection different analyses for the Door Status Control main function are performed to evaluate different design decisions and their influence on dependability and cost. By means of the dependability evaluation model, simulations are performed to evaluate: (1) redundancy strategies; (2) reconfiguration strategies; and (3) validity of the hypothesis of the ideal behaviour of fault detection, reconfiguration and communication.

Dependability Evaluation Model

According to the Dependability Evaluation Modelling approach, subfunction's implementations are characterized with its constituent resources' failure rates. For the Door Open Detection (DOD) subfunction (cf. Table 3.14, implementations #7 and #8), its implementation failures are specified as follows⁹ (see Equation 4.1 for more information about the λ notation):

$$\begin{aligned}\mathcal{F}_{\text{DoorOpenDetection}_1} &= \mathbf{OR}(\lambda_{\text{PU}_{\text{DSC_A}}}, \lambda_{\text{OpenSensor}}) \\ \mathcal{F}_{\text{DoorOpenDetection}_2} &= \mathbf{OR}(\lambda_{\text{PU}_{\text{Cam}}}, \lambda_{\text{Camera}}, \lambda_{\text{SW}_{\text{OpenDet}}}, \lambda_{\text{Comm}})\end{aligned}$$

The same equation holds for the failure characterizations of the omission failures of: fault detection of the door open detection ($\mathcal{F}_{\text{FD_DoorOpenDetection}_i \text{ O}}$ - #9), reconfiguration of the door open detection ($\mathcal{F}_{\text{R_DoorOpenDetection}_i \text{ O}}$ - #10, $\mathcal{F}_{\text{R_DoorOpenDetection}_2 \text{ O}}$ - #11), and fault detection of the door open detection's reconfiguration ($\mathcal{F}_{\text{FD_R_DoorOpenDetection}_1 \text{ O}}$ - #12; $\mathcal{F}_{\text{FD_R_DoorOpenDetection}_2 \text{ O}}$ - #13) subfunctions implementations:

$$\begin{aligned}\mathcal{F}_{\text{FD_DoorOpenDetection}_1 \text{ O}} &= \mathbf{OR}(\lambda_{\text{PU}_{\text{DSC_A}}}, \lambda_{\text{SW}_{\text{FD_DOD}}}, \lambda_{\text{Comm}}) \\ \mathcal{F}_{\text{R_DoorOpenDetection}_1 \text{ O}} &= \mathbf{OR}(\lambda_{\text{PU}_{\text{DSC_A}}}, \lambda_{\text{SW}_{\text{R_DOD}}}) \\ \mathcal{F}_{\text{R_DoorOpenDetection}_2 \text{ O}} &= \mathbf{OR}(\lambda_{\text{PU}_{\text{Cam}}}, \lambda_{\text{SW}_{\text{R_DOD}}}, \lambda_{\text{Comm}}) \\ \mathcal{F}_{\text{FD_R_DoorOpenDetection}_1 \text{ O}} &= \mathbf{OR}(\lambda_{\text{PU}_{\text{Cam}}}, \lambda_{\text{SW}_{\text{FD_R_DOD}}}, \lambda_{\text{Comm}}) \\ \mathcal{F}_{\text{FD_R_DoorOpenDetection}_2 \text{ O}} &= \mathbf{OR}(\lambda_{\text{PU}_{\text{DSC_A}}}, \lambda_{\text{SW}_{\text{FD_R_DOD}}}, \lambda_{\text{Comm}})\end{aligned}$$

Accordingly, the false positive failures will be characterized with their characterizing failure distribution and corresponding parameters: $\mathcal{F}_{\text{FD_DoorOpenDetection FP}} = \lambda_{\text{FD_DOD FP}}$, $\mathcal{F}_{\text{FD_R_DoorOpenDetection}_i \text{ FP}} = \lambda_{\text{FD_R_DoorOpenDetection}_i \text{ FP}}$ $i=\{1,2\}$.

The failure of the door open detection subfunction will be characterized according to the following equation:

$$\mathcal{F}_{\text{DoorOpenDetection}} = \mathbf{OR}(\mathcal{F}_{\text{All Impl_DOD}}, \mathcal{F}_{\text{Unresolved_DOD}}, \mathcal{F}_{\text{Dependencies_DOD}})$$

The $\mathcal{F}_{\text{All Impl_DOD}}$ event will happen when each implementation fails or is detected as

⁹For the sake of simplification we will include in λ_{Comm} failure rates of all the communication networks and interconnecting gateway device.

failed:

$$\mathcal{F}_{\text{All Impl.}_DOD} = \mathbf{AND}(\mathcal{F}_{\text{DoorOpenDetection}_1 \text{ FP}}, \mathcal{F}_{\text{DoorOpenDetection}_2 \text{ FP}})$$

where $\mathcal{F}_{\text{DoorOpenDetection}_i \text{ FP}} = \mathbf{OR}(\mathcal{F}_{\text{DoorOpenDetection}_i}, \lambda_{\text{FD_DoorOpenDetection}_i \text{ FP}})$; $i = \{1, 2\}$.

Since the door open detection subfunction has 2 implementations, the failure unresolved event will take into account the failure unresolved situation of the first implementation:

$$\mathcal{F}_{\text{Unr. Impl.}_DOD} = \mathbf{OR}(\mathcal{F}_{\text{R Seq.}_1 \text{ DoorOpenDetection}}, \mathcal{F}_{\text{FD Seq.}_1 \text{ DoorOpenDetection}})$$

The reconfiguration sequence failure and fault detection sequence failure for the first implementation of the door open detection subfunction are defined as follows:

$$\mathcal{F}_{\text{R Seq.}_1 \text{ DoorOpenDetection}} = \mathbf{PAND}(\mathcal{F}_{\text{R_DoorOpenDetection}}, \mathcal{F}_{\text{DoorOpenDetection}_1 \text{ FP}})$$

The reconfiguration failure $\mathcal{F}_{\text{R_DOD}}$ is developed as follows:

$$\mathcal{F}_{\text{R_DOD}} = \mathbf{OR}(\mathcal{F}_{\text{All R Impl.}_\text{DoorOpenDetection}}, \mathcal{F}_{\text{R Unresolved}_\text{DoorOpenDetection}})$$

where,

$$\begin{aligned} \mathcal{F}_{\text{All R Impl.}_\text{DoorOpenDetection}} &= \mathbf{AND}(\mathcal{F}_{\text{R_DoorOpenDetection}_1 \text{ O/FP}}, \mathcal{F}_{\text{R_DoorOpenDetection}_2 \text{ O/FP}}) \\ \mathcal{F}_{\text{R_DoorOpenDetection}_i \text{ O/FP}} &= \mathbf{OR}(\mathcal{F}_{\text{R_DoorOpenDetection}_i \text{ O}}, \lambda_{\text{R_DOD}_i \text{ FP}}) \\ \mathcal{F}_{\text{R Unresolved}_\text{DoorOpenDetection}} &= \mathbf{AND}(\mathcal{F}_{\text{FD_R_DoorOpenDetection}_1 \text{ O}}, \mathcal{F}_{\text{FD_R_DoorOpenDetection}_2 \text{ O}}) \end{aligned}$$

The fault detection sequence failure for the door open detection subfunction is defined as follows:

$$\mathcal{F}_{\text{FD Seq.}_\text{DoorOpenDetection}_1} = \mathbf{PAND}(\mathcal{F}_{\text{FD_DoorOpenDetection}}, \mathcal{F}_{\text{DoorOpenDetection}_1})$$

The fault detection failure of the door open detection $\mathcal{F}_{\text{FD_DoorOpenDetection}}$ depends on the operation of the destination subfunction (SF_{DEST}), because the FD implementation is located at the same PU:

$$\mathcal{F}_{\text{FD_DoorOpenDetection}} = \mathcal{F}_{\text{FD_Dest}_1}$$

The destination subfunction is the Door Control Algorithm (DCA) subfunction (implementation #23 in Table 3.14):

$$\mathcal{F}_{\text{FD_Dest}_1} = \mathbf{OR}(\mathcal{F}_{\text{DoorControlAlgorithm}_1}, \mathcal{F}_{\text{FD_DoorOpenDetection}_1 \text{ O}})$$

where,

$$\mathcal{F}_{\text{DoorControlAlgorithm}_1} = \mathbf{OR}(\lambda_{\text{PU}_{\text{DSC_A}}}, \lambda_{\text{SW_CL}}, \lambda_{\text{Comm}})$$

Note that door control subfunction's dependencies are not taken into account deliberately to avoid creating logical loops. At this level, we consider only the implementation failure itself, and when characterizing the failure of the door control subfunction ($\mathcal{F}_{\text{DoorControlAlgorithm}}$) its dependencies will be considered.

There is no input dependency for the door open detection subfunction, because it is an input subfunction and therefore, it does not require to receive data from another subfunction ($\mathcal{F}_{\text{Dependencies_DOD}} = 0$).

The failure characterization of the Door Closed Detection (DCD) subfunction failure (\mathcal{F}_{DCD}) follows exactly the same process accounting for its respective resources' failures.

The remainder of input subfunctions (Enable Door Passenger - EDP #3, Door Close Command - DCC #4, Door Open Command - DOC #5, #6, Door Velocity - DV #21 and Obstacle Detection - OD #22) do not have redundancies and therefore, their failure characterization is directly obtained through the failure characterization of the implementation's constituent resources:

$$\begin{aligned} \mathcal{F}_{\text{EnableDoorPassenger}} &= \mathcal{F}_{\text{EnableDoorPassenger}_1} = \mathbf{OR}(\lambda_{\text{EDD}}, \text{PU}_{\text{Driver}}, \lambda_{\text{EnableButton}_{\text{Driver}}}, \lambda_{\text{Comm}}) \\ \mathcal{F}_{\text{DoorCloseCommand}} &= \mathcal{F}_{\text{DoorCloseCommand}_1} = \mathbf{OR}(\lambda_{\text{PU}_{\text{Driver}}}, \lambda_{\text{CloseButton}_{\text{Driver}}}) \\ \mathcal{F}_{\text{DoorVelocity}} &= \mathcal{F}_{\text{DoorVelocity}_1} = \mathbf{OR}(\lambda_{\text{PU}_{\text{DSC_A}}}, \lambda_{\text{VelocitySensor}}) \\ \mathcal{F}_{\text{ObstacleDetection}} &= \mathcal{F}_{\text{ObstacleDetection}_1} = \mathbf{OR}(\lambda_{\text{PU}_{\text{DSC_A}}}, \lambda_{\text{ObstacleSensor}}) \end{aligned}$$

However, note that door open command has two 2 implementations which operate as

active redundancies:

$$\begin{aligned}\mathcal{F}_{\text{DoorOpenCommand}_1} &= \mathbf{OR}(\lambda_{\text{PU}_{\text{Driver}}}, \lambda_{\text{OpenButton}_{\text{Driver}}}) \\ \mathcal{F}_{\text{DoorOpenCommand}_2} &= \mathbf{OR}(\lambda_{\text{PU}_{\text{DSC_A}}}, \lambda_{\text{OpenButton}_{\text{Passenger}}}) \\ \mathcal{F}_{\text{DoorOpenCommand}} &= \mathbf{AND}(\mathcal{F}_{\text{DoorOpenCommand}_1}, \mathcal{F}_{\text{DoorOpenCommand}_2})\end{aligned}$$

As for the door control algorithm, there are no implementation redundancies, but there exist input dependencies. Therefore, its failure expression is as follows:

$$\mathcal{F}_{\text{DoorControlAlgorithm}} = \mathbf{OR}(\mathcal{F}_{\text{All Impl.}_{\text{DoorControlAlgorithm}}}, \mathcal{F}_{\text{Dependencies}_{\text{DoorControlAlgorithm}}})$$

where,

$$\begin{aligned}\mathcal{F}_{\text{All Impl.}_{\text{DoorControlAlgorithm}}} &= \mathcal{F}_{\text{DoorControlAlgorithm}_1} \\ \mathcal{F}_{\text{Dependencies}_{\text{DoorControlAlgorithm}}} &= \mathcal{F}_{\text{Dep. C_CL}} \\ \mathcal{F}_{\text{Dep. C_CL}} &= \mathbf{AND}(\mathcal{W}_{\text{C_CL}}, \mathcal{F}_{\text{I_CL}}) \\ \mathcal{W}_{\text{C_CL}} &= \mathbf{NOT}(\mathcal{F}_{\text{DoorControlAlgorithm}_1}) \\ \mathcal{F}_{\text{I_CL}} &= \mathbf{OR}(\mathcal{F}_{\text{EDP}}, \mathcal{F}_{\text{DCC}}, \mathcal{F}_{\text{DOC}}, \mathcal{F}_{\text{DoorOpenDetection}}, \mathcal{F}_{\text{DCD}}, \mathcal{F}_{\text{DV}}, \mathcal{F}_{\text{OD}})\end{aligned}$$

Therefore, after simplification¹⁰, the door control algorithm subfunction's failure is specified as follows:

$$\mathcal{F}_{\text{DCA}} = \mathbf{OR}(\mathcal{F}_{\text{DCA}_1}, \mathcal{F}_{\text{EDP}}, \mathcal{F}_{\text{DCC}}, \mathcal{F}_{\text{DOC}}, \mathcal{F}_{\text{DOD}}, \mathcal{F}_{\text{DCD}}, \mathcal{F}_{\text{DV}}, \mathcal{F}_{\text{OD}})$$

Finally, the failure of the door manipulation (DM) subfunction ($\mathcal{F}_{\text{DoorManipulation}}$) and accordingly, the failure of the Door Status Control main function is specified as follows:

$$\mathcal{F}_{\text{DoorManipulation}} = \mathbf{OR}(\mathcal{F}_{\text{All Impl.}_{\text{DM}}}, \mathcal{F}_{\text{Unresolved}_{\text{DM}}}, \mathcal{F}_{\text{Dependencies}_{\text{DM}}})$$

Note that the door manipulation subfunction has one implementation (#24), therefore: $\mathcal{F}_{\text{All Impl.}_{\text{DoorManipulation}}} = \mathcal{F}_{\text{DoorManipulation}_1}$ and $\mathcal{F}_{\text{Unresolved}_{\text{DoorManipulation}}} = 0$.

¹⁰ $A + \overline{A}.B = A + B$

$$\mathcal{F}_{\text{DoorManipulation}} = \mathbf{OR}(\mathcal{F}_{\text{DoorManipulation}_1}, \mathcal{F}_{\text{DoorControlAlgorithm}})$$

$$\mathcal{F}_{\text{DoorManipulation}_1} = \mathbf{OR}(\lambda_{\text{PU}_{\text{DSC_A}}}, \lambda_{\text{Motor}})$$

Redundancy Strategies

For simplicity, 2 heterogeneous redundancies have been considered in the *extended HW/SW architecture* displayed in Table 3.14 (door open detection, door closed detection). In order to add further optimization possibilities (and architecture combinations) to the *extended HW/SW architecture* all possible heterogeneous redundancies have been included. Therefore, within the design considerations we will include homogeneous and heterogeneous redundancies for obstacle detection and door velocity subfunctions, apart from the previously considered door open detection and door closed detection subfunctions' heterogeneous redundancies.

Figure 4.10 shows relative cost and failure probability of Door Status Control main function's alternative configurations with respect to the Door Status Control configuration without redundancies described in the functional model at Table 3.12. Among the 4 input subfunctions with heterogeneous redundancies (Door Open Detection - DOD, Door Closed Detection - DCD, Obstacle Detection - OD, and Door Velocity - DV), as Table 4.7 displays, alternative *extended HW/SW architectures* are analysed adding one additional heterogeneous redundancy and/or homogeneous redundancy to each subfunction using the reconfiguration strategy *2R Centralised* described in Table 4.8.

Table 4.7: Door Status Control Configurations with Alternative Redundancy Strategies

ID	Configuration
#1	No redundancies (cf. Table 3.12)
#2	4 Heterogeneous redundancies
#3	4 Homogeneous redundancies
#4	3 Heterogeneous redundancies: DCD, DOD, DV; 1 homogeneous redundancy: OD
#5	2 Heterogeneous redundancies: DCD, DOD; 2 homogeneous redundancies: OD, DV
#6	1 Heterogeneous redundancy: DCD; 3 homogeneous redundancies: OD, DV, DOD

In all the configurations displayed in Table 4.7, homogeneous redundancies are created by replicating the correspondent subfunction implementation’s sensor and connecting them to the existing PU_{DSC_A} operating as active redundancy.

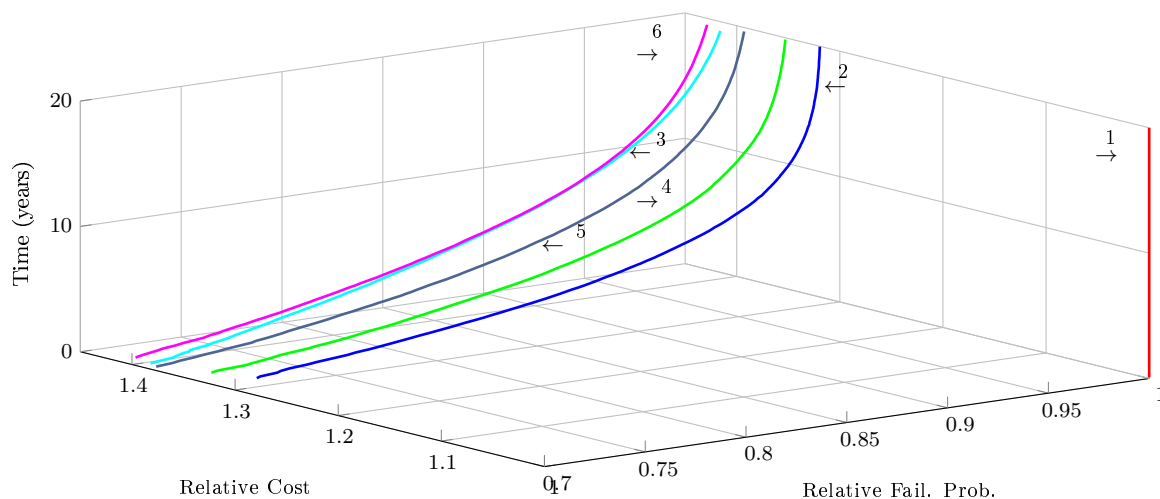


Figure 4.10: Relative Failure Probability & Cost of Alternative Door Status Control Main Function’s Configurations for the Train.Car₁.Zone_A.Door (10^6 iterations)

As Figure 4.10 depicts, heterogeneous redundancies are more economical than homogeneous redundancies, nevertheless, their drawback is that it is necessary to add further mechanisms (SW) to make implementations compatible, which leads to having slightly worse reliability than homogeneous redundancies due to the increased failure sources.

To analyse further differences between homogeneous redundancies and heterogeneous redundancies, we calculate the contribution of the door open detection subfunction failure on the main function failure (failure criticality index - cf. Section 4.3). At the same time, the uncertainty of the failure rate data (cf. Section 4.4) of the open detection subfunction software (SW_{Det}) has been taken into account. Figure 4.11 and Figure 4.12 show the distribution of the failure criticality index values of door open detection subfunction’s redundancy components with $\lambda_{SW_Det} = [0.001-0.1]$.

From Figure 4.11 and Figure 4.12 it is clear that the reuse of hardware components adds bottlenecks to the system design resulting in a worse \mathcal{FCI} value than distributing tasks among different components: in the heterogeneous redundancy configuration the camera is connected to one PU and the original sensor is in another PU (cf. Figure 4.11), while in the homogeneous redundancy configuration redundant sensors are connected to

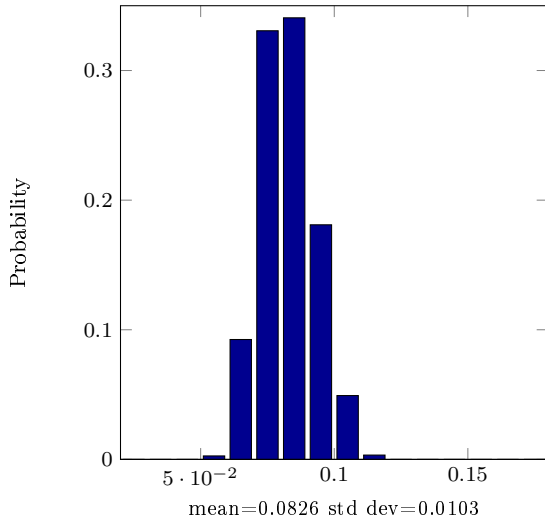


Figure 4.11: FCI_{DOD} - Heterogeneous Redundancy (10^6 iterations)

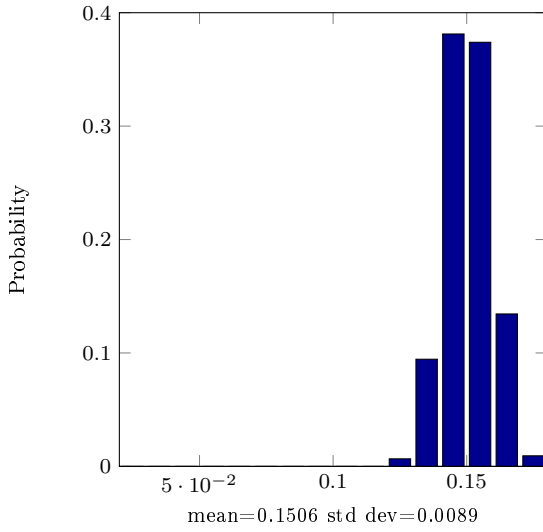


Figure 4.12: FCI_{DOD} - Homogeneous Redundancy (10^6 iterations)

the same PU (cf. Figure 4.12), which explains why heterogeneous implementations are less critical in this case.

Reconfiguration Strategies

To analyse the influence of the number and distribution of reconfiguration implementations on system dependability, this nomenclature is adopted: SF_i refers to the i -th implementation of the subfunction (e.g., R_DOD_1 designates the first implementation of the door open detection's reconfiguration subfunction) and 1R, 2R and 3R identify the number of reconfiguration implementations.

Based on the system architecture comprised of 4 heterogeneous redundancies, alternative reconfiguration strategies have been tested with different failure rate values of health management SW components (λ_{SW_HM}): SW_FD, SW_R and SW_FD_R. The failure rates of these software resources have been modified altogether to highlight the influence of reconfiguration implementations on system unreliability at the $T = 10$ years time instant with 10^6 Monte Carlo trials.

From Table 4.8 two main patterns are identified: the greater the λ_{SW_HM} and number of reconfiguration redundancies, the better the reliability of distributed reconfigurations. The unreliability of centralised reconfigurations confirms that the introduction of addi-

Table 4.8: Door Status Control Failure Probability for Reconfiguration Distribution Strategies (T=10 years)

Configuration	Reconfiguration Implementation Distributions	DSC Fail. Prob.		
		λ_{SW_HM} =0.05	λ_{SW_HM} =0.15	λ_{SW_HM} =0.25
1R Centralised	$\mathbf{PU}_1(R_DOD_1, R_DCD_1, R_OD_1, R_DV_1)$	0.856	0.887	0.902
1R Distributed	$\mathbf{PU}_1(R_DOD_1); \mathbf{PU}_2(R_DCD_1); \mathbf{PU}_3(R_OD_1); \mathbf{PU}_4(R_DV_1)$	0.867	0.892	0.904
2R Centralised	$\mathbf{PU}_1(R_DOD_1, R_DCD_1, R_OD_1, R_DV_1); \mathbf{PU}_2(R_DOD_2, R_DCD_2, R_OD_2, R_DV_2)$	0.850	0.888	0.905
2R Distributed	$\mathbf{PU}_1(R_DOD_1, R_DCD_2); \mathbf{PU}_2(R_DOD_2, R_DCD_1); \mathbf{PU}_3(R_OD_1, R_DV_2); \mathbf{PU}_4(R_OD_2, R_DV_1)$	0.853	0.888	0.905
3R Centralised	$\mathbf{PU}_1(R_DOD_1, R_DCD_1, R_OD_1, R_DV_1); \mathbf{PU}_2(R_DOD_2, R_DCD_2, R_OD_2, R_DV_2); \mathbf{PU}_3(R_DOD_3, R_DCD_3, R_OD_3, R_DV_3)$	0.838	0.874	0.897
3R Distributed	$\mathbf{PU}_1(R_DOD_1, R_DCD_2, R_OD_3); \mathbf{PU}_2(R_DOD_2, R_DCD_1, R_DV_3); \mathbf{PU}_3(R_DOD_3, R_OD_1, R_DV_2); \mathbf{PU}_4(R_DCD_3, R_OD_2, R_DV_1)$	0.839	0.875	0.897

tional components increase system failure sources. However, with the increase of the failure rate values and reconfiguration's redundancies, system's common cause failures gain importance and distributed implementations perform better than configurations with system bottlenecks.

Interestingly, we come up with a "threshold" failure probability, where from that point on, the distribution of reconfiguration strategies have no impact on the reliability of the system architecture. The "threshold" failure probability decreases as the number of reconfiguration's redundancy implementations increases (see grey cells in Table 4.8). This should be studied further, but it seems logical that the higher the unreliability of the reconfiguration implementations, the impact of the reconfiguration strategies becomes less important.

Influence of Health Management Implementations

To validate the feasibility of the assumption of the ideal (non-faulty) behaviour of fault detection, reconfiguration and communication implementations, we evaluate their influence on system's dependability under different assumptions.

Taking the configuration (2) of Figure 4.10 as the *reference* configuration, Figure 4.13 depicts the results of different architectures to test the feasibility of the hypotheses about the ideal behaviour of fault detection, reconfiguration and communication implementations. The outcome failure probability of different configurations has been normalized with respect to the *reference* configuration, in which the behaviour of the fault detection, reconfiguration and communication implementations have been considered with their respective failure characterization.

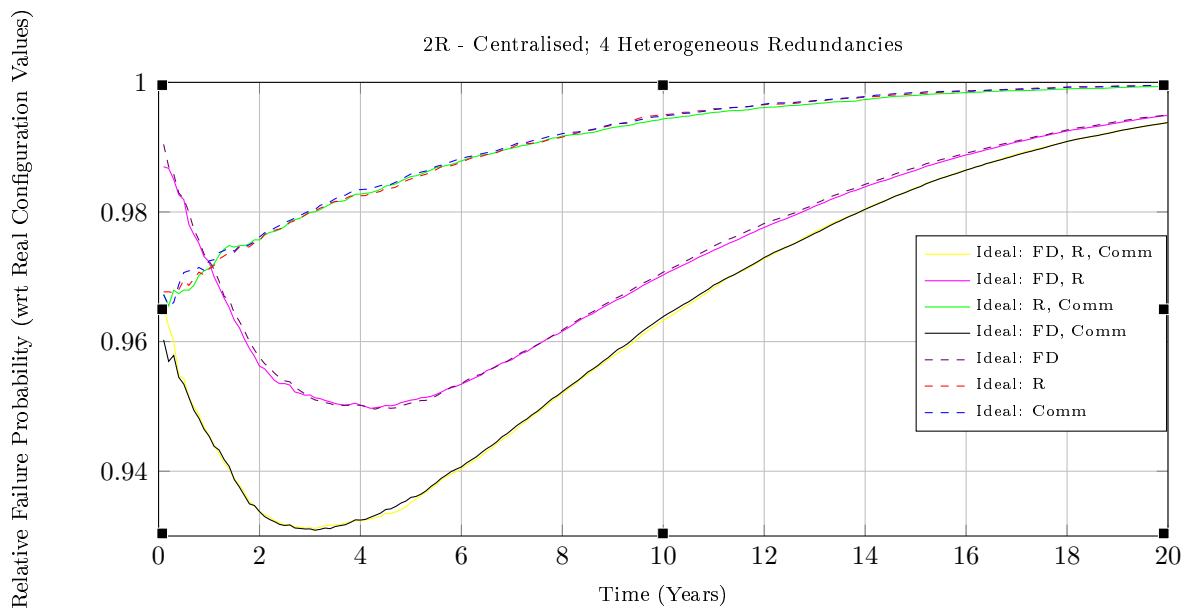


Figure 4.13: Door Status Control: Ideal Configurations Relative Failure Probabilities w.r.t. Reference Configuration (10^6 iterations)

As Figure 4.13 depicts, there is a 7% maximum difference between the ideal and the reference configurations in which the fault detection, reconfiguration and communication implementations are assumed perfectly reliable (cf. yellow line). Besides, the influence of the failure behaviour of the fault detection is also noticeable (dashed purple line). In this specific case, this issue is caused by the lack of redundancy implementations for the fault detection subfunction.

To further evaluate the influence of the fault detection and reconfiguration subfunction failures on system unreliability, failure criticality index evaluations have been performed for the configurations depicted in Figure 4.10: 4 heterogeneous redundancy and 4 homogeneous redundancy configurations. For the homogeneous redundancy configuration (configuration #3 in Table 4.7), 2 alternative arrangements have been tested: connect explicit homogeneous sensors to the same PU or connect explicit homogeneous sensors to different PUs. Table 4.9 displays the influence of the failure of fault detection and reconfiguration subfunctions on different Redundancy Strategies (RS).

Table 4.9: $\mathcal{FCI}_{\mathcal{F}_{FD_SF}}$ and $\mathcal{FCI}_{\mathcal{F}_{R_SF}}$ using Different Redundancy Strategies (10^6 iterations)

RS	$\mathcal{FCI}_{\mathcal{F}_{FD_DOD}}$	$\mathcal{FCI}_{\mathcal{F}_{R_DOD}}$	$\mathcal{FCI}_{\mathcal{F}_{FD_DCD}}$	$\mathcal{FCI}_{\mathcal{F}_{R_DCD}}$	$\mathcal{FCI}_{\mathcal{F}_{FD_OD}}$	$\mathcal{FCI}_{\mathcal{F}_{R_OD}}$	$\mathcal{FCI}_{\mathcal{F}_{FD_DV}}$	$\mathcal{FCI}_{\mathcal{F}_{R_DV}}$
A	0.1520	0.1367	0.1524	0.1374	0.1520	0.1372	0.1563	0.1416
B	0.2265	0.1949	0.2267	0.1956	0.2265	0.1954	0.2362	0.1999
C	0.1826	0.1623	0.1832	0.1632	0.1825	0.1627	0.1863	0.1674

A: 4 Homogeneous Redundancies connected to different explicitly added 4 PUs

B: 4 Homogeneous Redundancies connected to the same existing PU_{DSC}

C: 4 Heterogeneous Redundancies

Supporting the statements from Figure 4.13, Table 4.9 displays that the FCI values of fault detection subfunction failures have higher criticality than reconfiguration subfunction failures. With respect to the influence of alternative redundancy arrangements on system's failure occurrence, Table 4.9 also describes how the influence on the top event's failure occurrence of fault detection and reconfiguration subfunctions increases when concentrating redundancies in the same PU.

To check the consistency of the data depicted in Figure 4.13, Table 4.10 displays the failure criticality index values of alternative subfunction failures under different assumptions: door control algorithm ($\mathcal{FCI}_{\mathcal{F}_{DCA}}$) and door open detection ($\mathcal{FCI}_{\mathcal{F}_{DOD}}$) as an example of input subfunction's failure influence. Besides, the failure influences of the fault detection sequence of the door open detection ($\mathcal{FCI}_{\mathcal{F}_{FD_DOD\ Seq.}}$) and reconfiguration sequence of the door open detection ($\mathcal{FCI}_{\mathcal{F}_{R_DOD\ Seq.}}$) on the system failure occurrence are also analysed.

Figure 4.13 and Table 4.10 agree on the results, so that the less critical (more reliable) architecture is the ideal configuration and the more unreliable the configuration with

Table 4.10: Failure Probabilities and FCI Values for Configurations under Different Assumptions (10^6 iterations)

Config.	$\mathcal{F}_{DM} @ T = 5$	$\mathcal{FCI}_{\mathcal{F}_{DCA}}$	$\mathcal{FCI}_{\mathcal{F}_{DOD}}$	$\mathcal{FCI}_{\mathcal{F}_{FD_DOD\ Seq.}}$	$\mathcal{FCI}_{\mathcal{F}_{R_DOD\ Seq.}}$
Ideal: FD, R, Comm	0.8724	0.9222	0.0953	0	0
Ideal: Comm, FD	0.873	0.9221	0.1016	0	0.0522
Ideal: FD, R	0.878	0.9236	0.0931	0	0
Ideal: FD	0.879	0.9237	0.0994	0	0.0542
Ideal: Comm, R	0.9007	0.9278	0.2123	0.1461	0
Ideal: Comm.	0.9011	0.9279	0.2119	0.1456	0.0798
Ideal: R	0.878	0.9278	0.2121	0.146	0
Reference	0.906	0.9291	0.2085	0.1456	0.0851

the real reference model. Furthermore, we see that the influence of the fault detection is the most considerable compared with fault detection and communication. Let us focus on the column $\mathcal{FCI}_{\mathcal{F}_{DCA}}$: while assuming ideal reconfiguration and communication implementations differs in 0.14% and 0.129% from the *reference* configuration's failure criticality index value respectively, assuming ideal fault detection implementation does make a 0.584% difference between ideal fault detection and *reference* configuration.

Let us now focus on the column $\mathcal{FCI}_{\mathcal{F}_{DOD}}$: we can see that the configuration which assumes ideal fault detection (and combinations thereof with ideal reconfiguration and/or ideal communication) implementation has the biggest difference with respect to the *reference* configuration. Note that the door open detection subfunction is one of the contributors to the top event occurrence, but not the only one, the remainder of input subfunctions, door control algorithm subfunction and the door manipulation implementation's resources also do influence to the top-event failure occurrence.

As for the analysis of the influence of the communication on the system failure probability, uncertainty analyses have been implemented. To this end, different interval

values have been assigned to the communication's failure rate and we have analysed its influence on the distribution of the top-event failure frequency at the time instant $T = 5$ (cf. Figure 4.14). The analyses have been performed on the configuration with 4 heterogeneous redundancies and 2 centralised reconfiguration implementations.

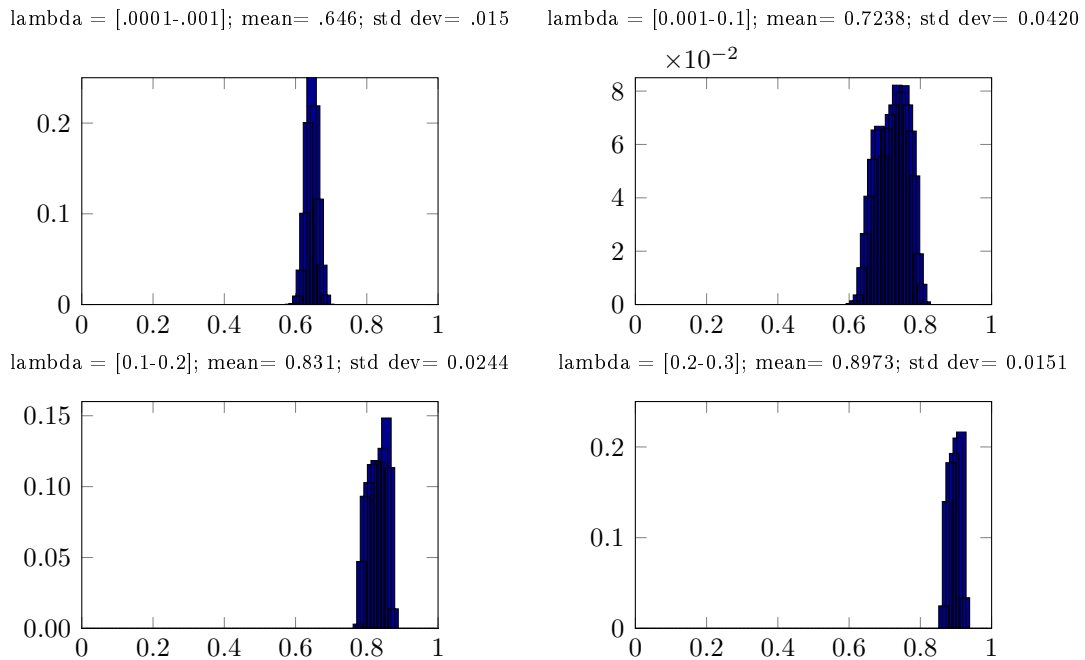


Figure 4.14: Failure Probability Distribution: 2R Centralised Heterogeneous Configuration - Communication's Failure Rate Intervals ($10^4 \times 5 \cdot 10^3$ iterations)

As Figure 4.14 confirms, an increase in the failure rate of the communication results in a worse system's failure probability. The shape of the system's failure probability distribution depends on the selected communication's failure rate interval.

4.7 Conclusions

In this chapter the algorithm and its implementation for the reliability assessment of the *extended HW/SW architecture* have been described. This approach makes possible the systematic evaluation of the influence on dependability and cost of redundancy strategies, reconfiguration strategies and the influence of health management and communication implementations on system failure probability. The probabilistic evaluation of the dependability evaluation model has been performed using the combination of

Dynamic Fault Tree and Component Fault Tree paradigms: Component Dynamic Fault Trees.

Besides, the outlined approach makes possible the evaluation of the influence of ideal/non-ideal fault detection, reconfiguration and communication implementations on system failure probability. To this end, the influence of these implementations on system failure probability has been taken into account and their contribution to the system failure occurrence has been evaluated using importance measurements.

Furthermore, in order to deal with the lack of exact failure data information of software resources as well as communication implementations, uncertainty analysis algorithms have been implemented within the Dependability Evaluation Modelling approach. The implementation enables the specification of interval failure rates (instead of single value data) and calculation of the failure probability distributions of top events failure probability occurrences.

As for the cost assessment of *extended HW/SW architectures* which use homogeneous or heterogeneous redundancies, the main difference remains in the software development cost. While homogeneous redundancies add an explicit hardware component to make possible the system reconfiguration, heterogeneous redundancies require additional (fit for purpose) software to reuse compatible implementations for further subfunctions. The cost of hardware resources is computed directly, but software implementations cost needs considering additional factors. Software cost can be divided into development and maintenance cost. Development of 2 different software implementations with similar characteristics is not quantified intuitively. In this dissertation we have grouped the development cost of similar SW resources considering their development cost once. Besides, we have also considered that the SW development costs will be paid off in X^{11} years (see Appendix E for the used failure rate and cost values).

All in all, the evaluation of which redundancy strategy is cheaper does not have only one answer. Depending on the type of heterogeneous redundancy strategy their costs also will be different. Generally speaking heterogeneous redundancies arising from natural compatibilities require less additional resources than heterogeneous redundancies arising from forced compatibilities. Therefore, depending on the type of heterogeneous redundancy strategy the comparison between homogeneous and

¹¹ $X=4$ years for calculation purposes.

heterogeneous redundancy strategies will be different.

In the case of infotainment functions, their failure occurrence does not pose critical issues for the system design. However, when considering safety-critical function failures their inability to perform can lead to the unavailability of the whole system. For instance, if a door of a train car fails (e.g., it is not possible to determine whether it is opened or closed) it is possible that the system requires stopping completely and the associated costs will increase considerably. In these cases, the use of heterogeneous redundancies to provide a compatible (and possibly degraded) functionality allows saving costs by exploiting already existing hardware resources.

The dependability analysis formalism presented in this chapter (Component Dynamic Fault Tree) is not able to evaluate the failure probability of the D3H2s compliant repairable HW/SW architectures. Although the extension of Component Dynamic Fault Trees to repairable systems is straightforward (i.e., considering repairable basic events), the CDFT approach in general and the priority-AND gate in particular are not able to handle complex repair policies. According to the priority-based reconfiguration process of the D3H2 methodology, when the failure of a subfunction's implementation is to be repaired, the implementation with the highest priority should be activated among the available redundancies for the failed subfunction. Therefore, the repair process may not be sequential as determined by the logic of the priority-AND gate. In order to grasp complex repair situations, more powerful formalisms needs to be considered as described in Chapter 5.

Furthermore, the cost calculation in this Chapter has been focused on the hardware, software, and communications cost. However, as it we will show in Chapter 5, the most penalizing cost is the one associated with system unavailability (downtime costs).

Dependability & Cost Analysis of Repairable Systems

In the D3H2 methodology, the dependability evaluation of the *extended HW/SW architectures* constituted by repairable resources sets new challenges. While in Chapter 4 only the order of failure was important, in this chapter the order of failure and the order of repair are addressed.

This chapter is organised as follows:

- Section 5.1 introduces the problem addressed in this chapter.
- Section 5.2 presents the Dependability Evaluation Modelling approach for repairable systems focusing on the evaluation algorithm and its implementation through the Stochastic Activity Networks (SAN) formalism.
- Section 5.4 applies the Dependability Evaluation Modelling approach for repairable systems to the running example of this dissertation.
- Section 5.5 closes this chapter with conclusions and prospects.

5.1 Introduction

In Chapter 4 we have constrained the Dependability Evaluation Modelling approach (DEM) (and the *extended HW/SW architecture*) with system implementations which use non-repairable resources. However, many of the current industrial systems are no longer characterized with non-repairable implementations. There exist mechanisms which make

possible the repair of system resources (either on-line or off-line) and improve the availability of the system. Shifting from non-repairable systems towards repairable systems introduces new challenges that the repairable DEM approach and its analysis paradigm must meet.

Namely, the characterization of the system's repair process governed by the priorities of the implementations is not trivial. In the D3H2 methodology, the repair behaviour of a system is characterized according to the reconfiguration table (see Chapter 3). The reconfiguration table determines alternative implementations (either homogeneous or heterogeneous) for the same subfunction and their corresponding priorities. Since implementations are assumed to be repairable, subfunction's repair process will be characterized according to the implementations priority. That is, the reconfiguration mechanism of the subfunction's implementation have to activate the implementation with the highest priority among the available spare implementations of the subfunction. This means that it does not necessarily have to follow a fixed sequence, e.g., assume that we have a subfunction with 4 implementations and currently the 3rd implementation is operative while first and second implementations are failed. If the first or the second implementation are repaired prior to the failure of the 3rd implementation, when the 3rd implementation fails the subfunction will be reconfigured to the 1st or the 2nd implementation instead of reconfiguring to the 4th implementation (cf. Figure 5.1 (a)).

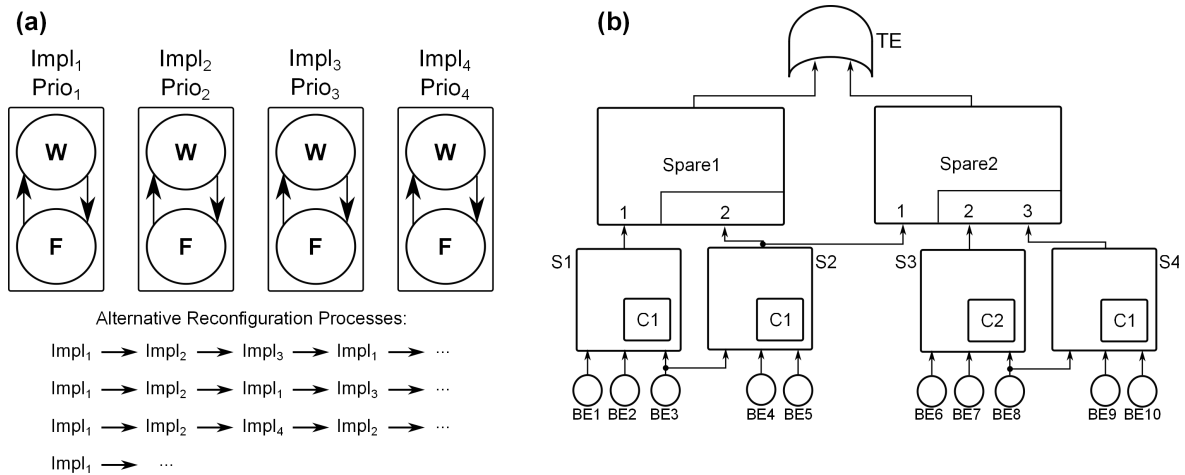


Figure 5.1: Challenges Emerging from Repairable Systems (a) Possible Reconfiguration Sequences (b) System Modelling through Dynamic Fault Tree's Spare Gates and Components

Although the manageability and ease of use of Component Dynamic Fault Tree formalism for non-repairable systems makes this paradigm suitable for the analysis of the DEM

approach, its application is limited to non-repairable *extended HW/SW architectures*. The repair process in the D3H2 methodology cannot be modelled using only sequential logic - as determined by the logic of the PAND gate. More powerful formalisms are required in order to manage the stated repair strategies. While it is possible to use Markov Chains to model such complex situations, in reality the use of pure Markov models is not feasible: the size and complexity of the resulting Markov model hampers understandability and maintainability of the system. The required size (number of states) to model such a complex (user defined) repair strategies would result in a unmanageable model.

The analysis of the *extended HW/SW architecture* which uses repairable resources opens the way to explore new system properties such as system availability and associated downtime costs. The downtime cost will provide the designer with an additional design indicator associated with the unavailability of the system. As we will see in the results (cf. Section 5.4), this cost will penalize more the less reliable architecture due to the increased downtime.

In order to deal with the stated properties and implement the compositional Dependability Evaluation Modelling approach for repairable systems, we have analysed existing formalisms looking for the following characteristics:

- Capability to model user-defined repair processes.
- Dynamic gates: capture the system failure logic accounting for time-ordered events.
- Capability to model repeated events and repeated components or subsystems.
- Component-based characterization.
- Support for any probability density function.

Figure 5.1 (b) shows an example of the systems that we analyse in this chapter using the concept of Dynamic Fault Tree's spare gates (see Subsection 2.3.1). Namely, systems with: (1) prioritized and repairable subsystems ($S1, S2, S3, S4$); (2) shared subsystems ($S2$); (3) repeated components among different subsystems ($C1$); and (4) repeated ($BE3, BE8$) and repairable basic events ($BE1, BE2, \dots, BE10$).

Furthermore, in addition to the assumptions adopted in Chapter 4 with respect to the

fixed architectural design decisions, throughout this chapter we consider that:

- The proactive/preventive maintenance is not applied and we will focus only on the reactive maintenance, i.e., the repair process starts only when a resource fails.
- We will deal with situations in which the repaired resources will be as good as new ones after the repair process without considering further degraded states.

5.2 Dependability Evaluation Modelling Approach for Repairable Systems

The compositional Dependability Evaluation Modelling approach for repairable systems enables to analyse the dependability of *extended HW/SW architectures* systematically.

5.2.1 Concepts and Notation

The objective of the DEM approach for repairable systems is the generic, systematic and complete failure and repair modelling of the *extended HW/SW architecture* to evaluate the dependability of alternative *extended HW/SW architectures*. The failure model for the DEM approach for repairable systems is the same as for the DEM approach for non-repairable systems (see Subsection 4.2.1).

With non-repairable resources (cf. Chapter 4) it is enough to assume that the implementations are reconfigured sequentially so that we know which implementation is active (working, operative) based on which implementations are failed. With repairable resources and implementations, it is necessary to check the status of all subfunction's implementations to know which implementation is **active** and accordingly determine system's failure situations (cf. Figure 5.1 (a)).

Let us define when the implementation i will be active: the implementation i will be active if (1) at the start of the system operation the implementation i has the highest priority among the implementations for the same subfunction; or (2) when a failure of the active implementation occurs (which is not the i -th implementation) and the

implementation i has the highest priority among the available implementations of the same subfunction.

Apart from the notation introduced in Table 4.1, we will also use additional notations to support the modelling of repairable systems as described in Table 5.1.

Table 5.1: Notation of Failure and Working Events II

<i>Notation</i>	<i>Failure Logic</i>
$\mathcal{F}_{\text{SF}_i \text{Active}}$	$[\text{SF}].[\text{Impl}_i]$ fail while active
$\mathcal{F}_{\text{SF}_i \text{ FP} \text{Active}}$	$[\text{SF}].[\text{Impl}_i]$ fail or FP while active = $\mathbf{OR}(\mathcal{F}_{\text{SF}_i \text{Active}}, \mathcal{F}_{\text{FD FP}})$
$\mathcal{F}_{\text{SF_Dest}_i \text{Active}}$	$[\text{SF_Dest}].[\text{Impl}_i]$ fail while active
$\mathcal{F}_{\text{FD}_i \text{ O} \text{Active}}$	$[\text{FD_SF}].[\text{Impl}_i]$ omission while active

The stochastic failure characterization of each resource is characterized by randomly sampling the failure and repair times according to their Cumulative Probability Distribution Functions (CDFs) along the system lifetime. The methodology supports any CDFs, but for the sake of simplicity without losing the generality of the approach, in subsequent probabilistic characterizations exponential failure distributions are assumed.

Hence, the failure characterization of system resources (\mathcal{F}_{Res}) is defined according to their failure rates (λ_{Res}) and repair rates (μ_{Res}). Assuming exponential failure and repair distributions, the failure characterization of system resources can be seen as Continuous Time Markov Chains with working and failed states. The transitions between these states are determined by failure rate (λ_{Res}) and repair rate (μ_{Res}).

The failure characterization of a SF's i -th implementation ($[\text{SF}].[\text{Impl}_i]$ *Failure*) comprised of N resources is specified as follows:

$$\mathcal{F}_{\text{SF}_i} = \mathbf{OR}(\mathcal{F}_{\text{Res}_1}, \mathcal{F}_{\text{Res}_2}, \dots, \mathcal{F}_{\text{Res}_N}) \quad (5.1)$$

The same equation holds for the characterizations of the omission failures of: fault detection subfunction (FD_SF - $\mathcal{F}_{\text{FD}_i \text{ O}}$), reconfiguration subfunction (R_SF - $\mathcal{F}_{\text{R}_i \text{ O}}$), and fault detection of the reconfiguration subfunction (FD_R_SF - $\mathcal{F}_{\text{FD_R}_i \text{ O}}$) implementations. Accordingly, the false positive failures of fault detection implementations ($\mathcal{F}_{\text{FD FP}}$

and $\mathcal{F}_{\text{FD_R}_i\text{ FP}}$) will be characterized with their characterizing failure and repair distributions and corresponding parameters (e.g., exponential distribution with $\lambda_{\text{FD FP}}$; $\mu_{\text{FD FP}}$ and $\lambda_{\text{FD_R}_i\text{ FP}}$ $\mu_{\text{FD_R}_i\text{ FP}}$ values). See Appendix E for failure and repair rate data values used in this dissertation.

5.2.2 Analysis Algorithm

The DEM approach for repairable systems determines the dependability evaluation algorithm. It defines the dynamic failure behaviour of systems which use fault detection and reconfiguration implementations covering all possible failure situations for the specified *extended HW/SW architectures*. It allows to evaluate the consequence of design decisions on system dependability systematically. Resulting equations characterize the failure of such systems compositionally so that the failure logic is kept clear for complex systems.

To this end, the DEM approach for repairable systems characterizes combinations of SF's implementation failures that prevent the *extended HW/SW architecture* from performing its intended SF¹². The SF will fail (\mathcal{F}_{SF}) when all implementations have failed ($\mathcal{F}_{\text{All Impl.}}$), an implementation fails and reconfiguration does not happen (failure unresolved, $\mathcal{F}_{\text{Unresolved}}$), or its input dependencies have failed ($\mathcal{F}_{\text{Dependencies}}$):

$$\mathcal{F}_{\text{SF}} = \mathbf{OR}(\mathcal{F}_{\text{All Impl.}}, \mathcal{F}_{\text{Unresolved}}, \mathcal{F}_{\text{Dependencies}}) \quad (5.2)$$

Assuming that we have N_{SF} implementations of the subfunction, the $\mathcal{F}_{\text{All Impl.}}$ event happens when each implementation fails or is detected as failed:

$$\mathcal{F}_{\text{All Impl.}} = \mathbf{AND}(\mathcal{F}_{\text{SF}_1\text{ FP}}, \dots, \mathcal{F}_{\text{SF}_{N_{\text{SF}}}\text{ FP}}) \quad (5.3)$$

The failure unresolved ($\mathcal{F}_{\text{Unresolved}}$) occurs when the active implementation fails and either the fault is not detected (failure undetected event) or the reconfiguration itself fails (reconfiguration failed event). For each implementation there are different failure

¹²The failure of any subfunction necessary for a main function provokes the immediate failure of a main function. Hence, from this point onwards, we will only consider the failure of a subfunction.

unresolved events ($\mathcal{F}_{\text{Unr. Imp}_i}$) because each implementation may have different failure probabilities:

$$\mathcal{F}_{\text{Unresolved}} = \mathbf{OR}(\mathcal{F}_{\text{Unr. Imp}_1}, \dots, \mathcal{F}_{\text{Unr. Imp}_{N_{SF}}}) \quad (5.4)$$

To define the failure unresolved event ($\mathcal{F}_{\text{Unr. Imp}_i}$) let us introduce two new events. The first event occurs when first the reconfiguration subfunction fails and then the i -th implementation of the subfunction fails when it is active (reconfiguration sequence failure, $\mathcal{F}_{\text{R Seq}_i}$):

$$\mathcal{F}_{\text{R Seq}_i} = \mathbf{PAND}(\mathcal{F}_{\text{R}}, \mathcal{F}_{\text{SF}_i \text{ FP} | \text{Active}}) \quad (5.5)$$

The second event occurs when first the fault detection of the SF fails and then the i -th implementation of the SF fails when it is active (fault detection sequence failure, $\mathcal{F}_{\text{FD Seq}_i}$):

$$\mathcal{F}_{\text{FD Seq}_i} = \mathbf{PAND}(\mathcal{F}_{\text{FD}}, \mathcal{F}_{\text{SF}_i | \text{Active}}) \quad (5.6)$$

Accordingly, the failure unresolved event of the i -th implementation ($\mathcal{F}_{\text{Unr. Imp}_i}$) occurs when either the fault detection sequence ($\mathcal{F}_{\text{FD Seq}_i}$) fails or the reconfiguration sequence ($\mathcal{F}_{\text{R Seq}_i}$) fails:

$$\mathcal{F}_{\text{Unr. Imp}_i} = \mathbf{OR}(\mathcal{F}_{\text{FD Seq}_i}, \mathcal{F}_{\text{R Seq}_i}) \quad (5.7)$$

Dependencies address Input (I) and Control (C) subfunctions influence on control and Output (O) subfunctions respectively. Control SF failure impacts directly the output subfunction failure (C→O); and the influence of input subfunction on control subfunction depends if the system's control configuration is operating in Closed Loop (C_CL) or Open Loop (C_OL):

$$\mathcal{F}_{\text{Dependencies}} = \mathbf{OR}(\mathcal{F}_{\text{Dep. C_CL}}, \mathcal{F}_{\text{Dep. C_OL}}) \quad (5.8)$$

Assuming that $\mathcal{W}_{C_X} = \mathbf{OR}(\mathcal{W}_{C_X_1}, \dots, \mathcal{W}_{C_X_{N_{\mathcal{W}}}})$ means that any of the $N_{\mathcal{W}}$ implementations of the C_X subfunction are working (where $X = \{CL, OL\}$), Equations in 5.9 describe the different input SFs that affect each control configuration (I_CL→C_CL, I_OL→C_OL). $\mathcal{F}_{\text{Dep. C_OL}}$ may not happen because the OL control generally does not have input dependencies:

$$\mathcal{F}_{\text{Dep. C_CL}} = \mathbf{AND}(\mathcal{W}_{C_CL}, \mathcal{F}_{I_CL}) \quad \mathcal{F}_{\text{Dep. C_OL}} = \mathbf{AND}(\mathcal{W}_{C_OL}, \mathcal{F}_{I_OL}) \quad (5.9)$$

The reconfiguration failure is a special subfunction and therefore \mathcal{F}_{R} is developed like Equation 5.2, except that there are no additional dependencies:

$$\mathcal{F}_{\text{R}} = \mathbf{OR}(\mathcal{F}_{\text{All R Impl.}}, \mathcal{F}_{\text{R Unresolved}}) \quad (5.10)$$

$\mathcal{F}_{\text{All R Impl.}}$ indicates the failure of all reconfiguration implementations, and $\mathcal{F}_{\text{R Unresolved}}$ designates the reconfiguration's failure unresolved condition. Assuming M reconfiguration implementations:

$$\mathcal{F}_{\text{All R Impl.}} = \mathbf{AND}(\mathcal{F}_{\text{R}_1 \text{ O/FP}}, \dots, \mathcal{F}_{\text{R}_M \text{ O/FP}}) \quad (5.11)$$

$\mathcal{F}_{\text{R Unresolved}}$ happens when M implementations of the FD_R_SF fail simultaneously and it is a direct consequence to our design choice: all reconfiguration's fault detection implementations (FD_R_SF) are active and homogeneous redundancies (heartbeat implementations):

$$\mathcal{F}_{\text{R Unresolved}} = \mathbf{AND}(\mathcal{F}_{\text{FD_R}_1}, \dots, \mathcal{F}_{\text{FD_R}_M}) \quad (5.12)$$

Accordingly, the false positive of the reconfiguration's fault detection occurs when all FD_R_SF implementations raise the false positive condition simultaneously. Although the system may operate correctly when a false positive occurs, it has to assume that the information provided by the fault detection is correct, since there is no mechanism to

detect the incorrect operation of fault detection.

The fault detection failure \mathcal{F}_{FD} depends on the operation of the destination subfunction (SF_{DEST}), because the FD implementation is located at the same PU. Hence, $\mathcal{F}_{\text{SF_DEST}}$ influences directly \mathcal{F}_{FD} .

When the FD implementation fails, the change of SF_{DEST} 's implementation determines its reconfiguration. We assume that the change of destination SF's implementation activates the corresponding FD implementation and the previous one is deactivated. Equation 5.13 describes the FD_SF failure case when FD_SF has K implementations:

$$\mathcal{F}_{\text{FD}} = \mathbf{OR}(\mathcal{F}_{\text{FD_Dest}_1 \mid \text{Active}}, \dots, \mathcal{F}_{\text{FD_Dest}_K \mid \text{Active}}) \quad (5.13)$$

The i -th fault detection implementation's failure while it is active ($\mathcal{F}_{\text{FD_Dest}_i \mid \text{Active}}$), expresses the following event: either i -th destination subfunction or the i -th fault detection implementation fail while they are active (remember that the i -th fault detection is located at the same PU as the SF_DEST_i implementation - see architectural design decisions and hypotheses at Section 4.1):

$$\mathcal{F}_{\text{FD_Dest}_i \mid \text{Active}} = \mathbf{OR}(\mathcal{F}_{\text{SF_DEST}_i \mid \text{Active}}, \mathcal{F}_{\text{FD}_i \text{ O} \mid \text{Active}}) \quad (5.14)$$

To avoid creating loops when evaluating system's dependability, we have considered that fault detection implementation's failure is determined by the destination subfunction's implementations failure without considering destination subfunction's input dependencies (cf. Equation 5.14). If dependencies are taken into account, they will create logical loops. Therefore, the influence of dependencies is taken into account at the "top" subfunction's failure level (cf. Equation 5.2). At this level, if any dependent subfunction fails, it leads directly to the failure occurrence of the subfunction.

5.2.3 Implementation

Stochastic Activity Networks (SAN) formalism meets all the design requirements (cf. Section 5.1) needed to model *extended HW/SW architectures* effectively and intuitively [Sanders02a]. Namely, SAN enables to model:

- (1) User defined repair priorities using input and output gates.
- (2) The behaviour of dynamic gates using places, activities and input and output gates.
- (3) Repeated events and components through the replicate/join formalism.
- (4) Any probability density function through simulations.

Stochastic Activity Networks

SAN formalism [Sanders02a] extends Petri Nets model by generalizing the stochastic relationships and introducing mechanisms to construct hierarchical models. SAN modelling primitives include places, activities, input gates, and output gates [Sanders12] (see Figure 5.2).







Standard Place	Extended Place	Input Gate	Output Gate	Instantaneous Activity	Timed Activity
					

Figure 5.2: Graphical Notation of SAN Elements

Places represent the state of the modelled system. Each place contains a certain number tokens defining the *marking* of the place: a *standard place* contains integer number of tokens, while *extended places* contain other data types than integers (e.g., floats, array).

There are two types of *activities*: (1) *instantaneous activities* represent system activities which complete in negligible amount of time; and (2) *timed activities* represent activities of the modelled system whose duration has an effect on the system performance. With timed activities the completion time can be a constant value or a random value. When the completion time is random its value has to be ruled by a probability distribution defining the time to fire the activity. Parameters of activities may be marking (token) dependant.

Activities *fire* based on the conditions defined over the marking of the net and their effect is to modify the *marking* of the places. The completion of an activity of any kind is enabled by a particular *marking* of a set of places. Assuming that there are neither input nor output gates, each activity has input and output arcs linked with its

input and output places respectively. The presence of at least one token in each input place enables the firing of the activity and removing the token from its input places and placing them in the output gates.

Associated with each activity is a *reactivation* function. This function defines the marking dependent conditions under which an activity is reactivated, that is, the activity is aborted and a new activity time is immediately obtained from the activity time distribution. The reactivation function consists of an activation predicate and reactivation predicate. An activity is reactivated at the moment of a marking change if (1) the reactivation predicate holds for the new marking; (2) the activity remains enabled; and (3) the activation predicate holds for the marking in which the activity was originally activated.

Each activity may have more than one *case* associated to it, which stands for a possible outcome of the activity. Each case corresponds to a certain effect of the completion of an activity and has a predetermined probability.

Another way to enable a certain activity consists of *input and output gates*. I/O gates make SAN formalism general and powerful enough to model complex real situations. They determine the marking of the net based on user-defined rules, which determine when an activity fires and its effect on the marking of the net.

Input gates control the enabling of activities and define the marking changes that will occur when an activity completes. A set of places are connected to the input gate and the input gate is connected to an activity characterizing the marking of the net based on two expressions:

- *Enabling predicate*: a boolean condition expressed in terms of the marking of the places connected to the gate; if such condition holds, then the activity connected to the gate is enabled.
- *Input Function*: the effect of the activity completion on the marking of the places connected to the gate.

An *output gate* is connected to an activity and a set of places and it defines the marking changes that will occur when an activity completes. It specifies the effect of activity completion on the marking of the places connected to the output gate. Output gates are defined only with an output function. The function defines the marking changes

that occur when the activity completes.

The *replicate/join* operators allow to model through a compositional tree structure different *atomic SAN models* linked in a unique component-based *composed model*. In the tree structure, atomic SAN models are linked together through *join* operators using the shared places between SAN models. The *replicate* operator constructs a number of identical copies of the SAN model through the *replicate* operator (same concept as Parametric Fault Trees [Codetta-Raiteri05]).

Therefore, the analyst can focus on specific characteristics of the system behaviour through fit-for-purpose atomic models and later join independently validated atomic models to obtain a more complex composed system model.

The performance measures are carried out through reward variables by choosing a specific solver to generate the solution. Reward functions are defined in order to retrieve a performance measurement over the specified model. There are two kind of reward functions (1) state reward functions, which are based on the marking of the net; and (2) impulse reward functions, which are based on the completion of the activities. The performance measurement is evaluated as the expected value of the reward function.

The modelling and analysis of SAN models is performed through the Möbius tool [Courtney04; Illinois14]. Please refer to [Sanders02a; Sanders12] for more information and formal definition of SAN formalism.

5.3 Cost Analysis

Apart from the hardware and software cost described in Section 4.5, **downtime costs** are also included when studying repairable systems to reflect the penalization incurred due to the system's unavailability.

In the specific case of railway systems, downtime cost is a critical factor which impacts negatively the overall economic budget. The downtime cost will be measured as the combination of: (1) number of travels lost while the train was stopped (*travels_lost*); (2) number of people in each travel (*people_travel*); and (3) average cost of a ticket per person (*ticket_cost*):

$$downtime_cost = travels_lost \times people_travel \times ticket_cost$$

$$travels_lost = \frac{travels}{hour} \times downtime$$

$$downtime = unavailability \times mission\ time$$

We will assume that we do not have to stop the whole train in order to fix a failure in a single car. Besides, for calculation purposes let us assume the following values (common values for a short-distance (< 50 km) train): $\frac{travels}{hour} = 2$; $people_travel = 20$; and $ticket_cost = 1 \text{ €}$. The *mission time* will be considered 30 years and we will evaluate the *unavailability* at $T = 30$ years time instant.

5.4 Results

Since the detailed dependability analysis of repairable Door Status Control and Fire Protection Control main functions require considering similar underlying concepts, in Subsection 5.4.1 we introduce the key concepts and models for the dependability analysis of *extended HW/SW architectures* using a simple example. Applying the concepts and models explained in Subsection 5.4.1, dependability and cost evaluations of the Fire Protection Control and Door Status Control main functions are examined in Subsection 5.4.2 and Subsection 5.4.3 respectively.

5.4.1 SAN Generic Models

Consider the hypothetical system displayed in Table 5.2 comprised of prioritized implementations each of them characterized by their constituting resources, in turn characterized with their corresponding failure and repair rates. This model is simple but representative enough to describe the main dependability modelling characteristics that are used to analyse more complex *extended HW/SW architectures*.

In the remainder of this subsection, we apply the equations described in the Dependability Evaluation Modelling approach for repairable systems (cf. Section 5.2) to the hypothetical *extended HW/SW architecture* displayed in Table 5.2 in a bottom-up manner. System's failure probability calculation is performed using the SAN formalism by

Table 5.2: Repairable HW/SW Architecture Example

MF	SF	Subfunction	Type	Implementation	Resources	Priority	#
MF	SF	SF	I	Impl1	Res1, Res2, Res3	1	1
		SF	I	Impl2	Res2, Res4, Res5	2	2
		FD_SF	FD	FD_Impl1	Res2, Res6, Res7	1	3
		R_SF	R	R_Impl1	Res2, Res7, Res8	1	4
		R_SF	R	R_Impl2	Res7, Res9, Res10	2	5
		FD_R_SF	FD_R	FD_R_Impl1	Res2, Res10, Res11	1	6
		FD_R_SF	FD_R	FD_R_Impl2	Res7, Res10, Res12	1	7
		ControlSF	C	C_Impl1	Res1, Res12, Res13	1	8
		OutputSF	O	O_Impl1	Res1, Res14, Res15	1	9

means of the Möbius tool.

Resources

Resources are the most basic models in the DEM approach. The failure characterization of *resources* (\mathcal{F}_{Res}) is defined according to their failure and repair rates.

In the SAN notation, we model *resources* with atomic models characterizing their failure and repair rates through activities. Figure 5.3 describes the characterization of the *resource Res1*: places *Res1_OK* and *Res1_KO* model working and failed states and activities *Res1_Fail* and *Res1_Repair* model failure and repair activities with their corresponding probabilistic distribution and parameters.

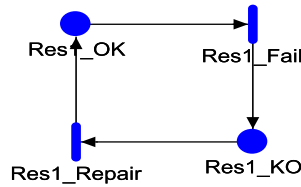


Figure 5.3: Atomic Model of Resources ($R01_Res1$)

In order to use a consistent nomenclature throughout this chapter, the models

of *resources* are denoted as $RX_Resource$ where X identifies the *resource* $X = \{1, 2, \dots, 15\}$ (cf. Table 5.2).

Implementations

The failure of each implementation is characterized according to the working or failure states of its constituting resources (cf. Equation 5.1).

There are two kind of implementations: *implementations without redundancies* and *implementations with redundancies*. In both cases, the implementations will be characterized with two interconnected models: an *atomic model* describing implementation's failure/repair logic; and a *composed model* which links: (1) the models of the implementation's resources describing their failure/repair logic (cf. Figure 5.3); with (2) the model of the implementation which describes its failure/repair logic (cf. Figure 5.4, Figure 5.6).

The implementation's models are named as $I\#_Implementation$ and their composed models are named as $top_I\#_Implementation$ where $\#$ identifies the implementation $\# = \{1, 2, \dots, 9\}$.

Implementations without redundancies are modelled with failed and repair events without the need to activate redundant implementations. Figure 5.4 presents the SAN model of the implementation #3 from Table 5.2. *Implementations without redundancies* are characterized with the following places:

- $I_FD_Impl1_Active$: implementation active place. The initial marking of this place will be 1.
- $I_FD_Impl1_KO$: implementation failed place. The input gate $Fail_FD_Impl1$ sets the marking of this place to 1 if any of its constituting resources (Res2, Res6, Res7) is failed, otherwise if all the resources are working the marking of this place will be zero.
- $Res2_KO$, $Res6_KO$, $Res7_KO$: these places indicate the failure of the implementation's constituent resources (see Figure 5.3).

Table 5.3 displays the failure and repair activities behaviour modelled through

Table 5.3: Activities in *I03_FD_Impl1*

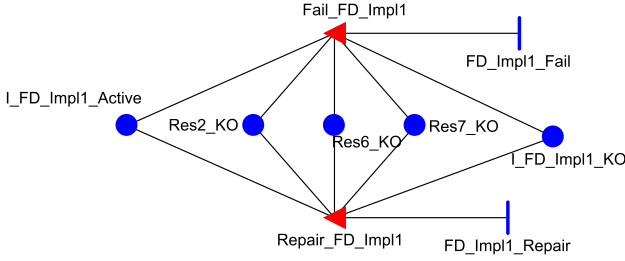


Figure 5.4: Atomic Model of *Implementations without Redundancies* (*I03_FD_Impl1*, #3)

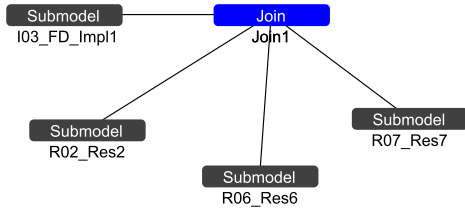


Figure 5.5: Composed Model of *Implementations without Redundancies* (*top_I03_FD_Impl1*)

Activity:	FD_Impl1_Fail
Time to complete:	Immediate
Input gate:	Fail_FD_Impl1
Input gate predicate:	$(m(I_FD_Impl1_KO)==0 \quad \&\& \\ (m(Res2_KO)==1 m(Res6_KO)==1 \\ m(Res7_KO)==1))$
Input gate function:	$m(I_FD_Impl1_Active)=0; \\ m(I_FD_Impl1_KO)=1;$
Activity:	FD_Impl1_Repair
Time to complete:	Immediate
Input gate:	Repair_FD_Impl1
Input gate predicate:	$(m(I_FD_Impl1_KO)==1 \quad \&\& \\ m(Res2_KO)==0 \quad \&\& \\ m(Res6_KO)==0 \quad \&\& \\ m(Res7_KO)==0)$
Input gate function:	$m(I_FD_Impl1_Active)=1; \\ m(I_FD_Impl1_KO)=0;$

Fail_FD_Impl1 and *Repair_FD_Impl1* input gates¹³ and Figure 5.5 displays the composed model, which links the models of resources (cf. Figure 5.3) and implementation (cf. Figure 5.4) using the join operator and shared places. Using the join operator selected places are shared among the models that contain this place. By means of shared places, repeated resources and repeated components are modelled.

Again for reading purposes, we will simplify the information shown in the following input gate tables. Specifically we will omit the linked **activity** which can be seen in the corresponding figure and we will also omit the **time to complete** because in all the cases studied throughout this chapter it is always *immediate*.

Implementations with redundancies require to (de)activate (or reconfigure) redundant implementations according to the implementations' states and priorities. Figure

¹³The function $m(x)$ denotes the marking of the place x .

5.6 depicts the SAN model of the implementation #1 from Table 5.2.

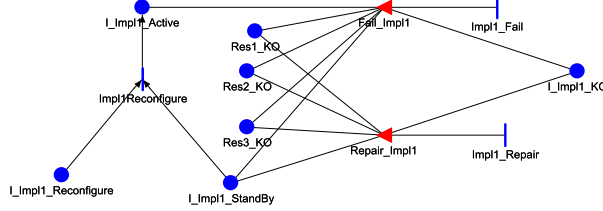


Figure 5.6: Atomic Model of the *Implementations with Redundancies (I01_Impl1, #1)*

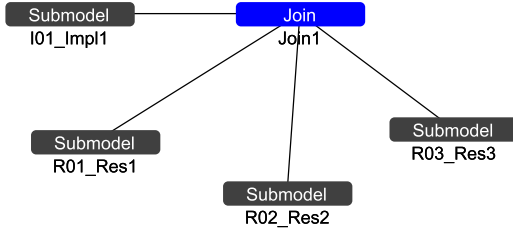


Figure 5.7: Composed Model of *Implementations with Redundancies (top_I01_Impl1)*

Table 5.4: Activities in *I01_Impl1*

Input gate:	Fail_Impl1
Input gate predicate:	$((m(\text{Res1_KO})==1 \parallel m(\text{Res2_KO})==1 \parallel m(\text{Res3_KO})==1) \&\& m(\text{I_Impl1_KO})==0)$
Input gate function:	if(m(I_Impl1_Active)==1) m(I_Impl1_Active)=0; if(m(I_Impl1_StandBy)==1) m(I_Impl1_StandBy)=0; m(I_Impl1_KO)=1;
Input gate:	Repair_Impl1
Input gate predicate:	$(m(\text{I_Impl1_KO})==1 \&\& m(\text{I_Impl1_StandBy})==0 \&\& m(\text{Res1_KO})==0 \&\& m(\text{Res2_KO})==0 \&\& m(\text{Res3_KO})==0)$
Input gate function:	m(I_Impl1_KO)=0; m(I_Impl1_StandBy)=1;

Replicated subfunction's implementations are characterized with the following places:

- *I_Impl1_Active*: implementation active place. If the implementation's priority is the highest, then the initial marking of this place will be 1, otherwise the initial marking of this place will be zero.
- *I_Impl1_KO*: implementation failed place. The input gate *Fail_Impl1* sets the marking of this place to 1 if any of its constituting resources fails (Res1, Res2, Res3). If all resources are operative the marking of this place will be zero.
- *I_Impl1_StandBy*: implementation standby place. If the priority of the implementation is the highest among the subfunction's implementations, the marking of this place will be zero. Otherwise, if the priority of the implementation is not the highest or if the implementation has been repaired after a failure the marking of this place will be 1.
- *I_Impl1_Reconfigure*: reconfiguration place. This place will be activated through the reconfiguration implementation model (see Figure 5.8). The marking of this

place is zero until the reconfiguration implementation logic decides to reconfigure an implementation and sets its marking to 1. Therefore, when this place is set to 1 and the marking of the *StandBy* place is one, the implementation will be activated immediately setting again the marking of the place *I_Impl1_Active* to one.

- *Res1_KO*, *Res2_KO*, *Res3_KO*: these places indicate the failure of the implementations constituent resources according to the logic described in Figure 5.3.

Note that an implementation may fail either when it is in active operation or in standby operation. Table 5.4 displays failure and repair characterizations of input gates for implementations with redundancies. The composed model of the *implementation with redundancies* is depicted in Figure 5.7. The difference with the *implementation without redundancies* is on higher levels when connecting composed models of implementations with their reconfiguration logic and failure logic.

The shared resources among implementations act as common cause failures for all the implementations which use the places of the resources as a part of the implementations failure characterization.

Reconfiguration's functional operation

When a implementation fails, the *reconfiguration* implementation has to activate an available redundant implementation taking into account implementation's priorities. In order to manage the marking of the *reconfiguration* places in the models of the implementations with redundancies (tagged with the suffix *Reconfigure* in the Figure 5.6, i.e., *I_Impl1_Reconfigure*) the model of the *reconfiguration* logic is created (cf. Figure 5.8).

The *reconfiguration* logic in Table 5.5 enables the priority-based reconfiguration. The *Reconfigure* input gate *reconfigures* subfunctions' implementations via shared places *I_Impl1_Reconfigure* (see Figure 5.6) and *I_Impl2_Reconfigure*. If the *reconfiguration* logic determines that an implementation should be reconfigured, it sets a token in its respective *Reconfigure* place. When there is a token in *I_Impl1_Reconfigure* or *I_Impl2_Reconfigure* place, the corresponding implementation (according to the model of the implementation with redundancies - see Figure 5.6) moves the token from the *StandBy* place to the *Active* place and it starts operating immediately. Table 5.5 displays the logic of the *Reconfigure* input gate.

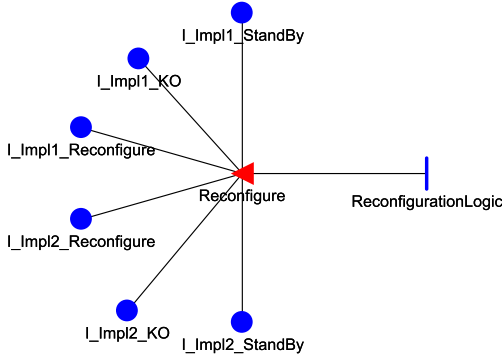


Figure 5.8: Atomic Model of the Reconfiguration Logic (*ReconfigurationLogic_SF*)

Table 5.5: Activities in *ReconfigurationLogic_SF*

Input gate:	Reconfigure
Input gate predicate:	$((m(I_Impl1_KO)==1 \ \&\&$ $m(I_Impl2_StandBy)==1) \ \ \$ $(m(I_Impl2_KO)==1 \ \&\&$ $m(I_Impl1_StandBy)==1)))$
Input gate function:	if($m(I_Impl1_StandBy)==1$) $m(I_Impl1_Reconfigure)=1;$ else if ($m(I_Impl2_StandBy)==1$) $m(I_Impl2_Reconfigure)=1;$

The input gate function in the *Reconfigure* input gate enables to reconfigure the subfunction's implementation to the available highest priority implementation. The models which characterize the *reconfiguration* logic are named as *ReconfigurationLogic_XX* where *XX* identifies the specific subfunction ($XX=\{SF, R_SF, FD_R_SF\}$).

Implementation fails while active

With non-repairable resources it is enough to assume that implementations are reconfigured sequentially and the logic for the system operation can be defined based on the failed implementations. However, with repairable resources it is necessary to keep track of which implementation is active to define the failure logic of system events (cf. Figure 5.1 (a)).

To determine which implementation is active, it is necessary to check the status of all subfunction's implementations. In the hypothetical example displayed in Table 5.2 there are two implementations for the subfunction *SF*: *Impl1* (#1) and *Impl2* (#2). *Impl1* fails while it is active when the first implementation fails ($m(I_Impl1_KO)=1$) and the second implementation is not active ($m(I_Impl2_Active)=0$).

The input gates *Impl1_FailActive* and *Impl1_NoFailActive* in Figure 5.9 implement the logic displayed in Table 5.6. The composed model depicted in Figure 5.10 links (1) composed models of the subfunction implementations (*top_I01_Impl1*, *top_I02_Impl2* - see Figure 5.6), (2) the reconfiguration logic to switch from one implementation to other (*ReconfigurationLogic_SF* - see Figure 5.8) and (3) *fail while active* failure logic

($F01_Impl1_FailActive$ - see Figure 5.9).

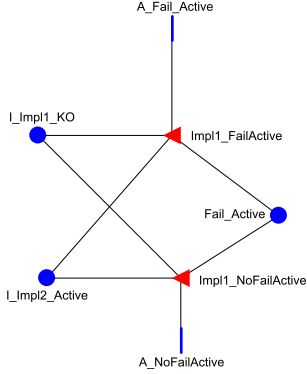


Figure 5.9: Atomic Model of the Fail while Active Logic ($F01_Impl1_FailActive$)

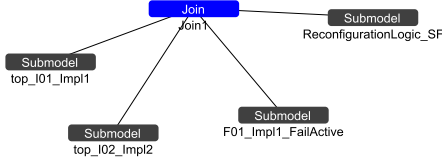


Figure 5.10: Composed Model of the Fail while Active Logic ($top_F01_Impl1_FailActive$)

Table 5.6: Activities in $F01_Impl1_FailActive$

Input gate:	Impl1_FailActive	
Input gate predicate:	$(m(I_Impl1_KO)==1$ $m(I_Impl2_Active)==0$ $m(Fail_Active)==0)$	$\&\&$ $\&\&$
Input gate function:	$m(Fail_Active)=1;$	
Input gate:	Impl1_NoFailActive	
Input gate predicate:	$(m(Fail_Active)==1 \&\&$ $m(I_Impl1_KO)==0$ $m(I_Impl2_Active)==1)$	\parallel
Input gate function:	$m(Fail_Active)=0;$	

The models of the failure events are named as Fnn_SF_FM where nn identifies the failure number, SF names the subfunction $SF=\{SF, R_SF, Impl1, Impl2, FD_R_SF, ControlSF, OutputSF\}$ and FM identifies the failure mode of the subfunction $FM=\{FailActive, AllFail, AllRFailed, RUnresolved, Failure, RF1, RF2, RF, FU1, FU2, FU\}$; where FU stands for failure undetected event (equivalent to the fault detection sequence failure event $\mathcal{F}_{FD\ Seq}$); and RF stands for reconfiguration failed event (equivalent to the reconfiguration sequence failure event $\mathcal{F}_{R\ Seq}$) see Table 5.7.

Table 5.7: Fault Detection and Reconfiguration Failure Events and Assigned Names

Event	Place name	Event	Place name
$\mathcal{F}_{FD\ Seq}$	FU	$\mathcal{F}_{R\ Seq}$	RF
$\mathcal{F}_{FD\ Seq.i}$	FU $_i$	$\mathcal{F}_{R\ Seq.i}$	RF $_i$

All implementations failed

All implementations failed event (cf. Equation 5.3) describes the situation in which all the implementations of a subfunction are failed at the same time (cf. Figure 5.11). The place *AllImplFailed* indicates the *all implementations failed* event and the failure logic is determined by the input gates *AllFailed* and *AllNoFailed* (cf. Table 5.8).

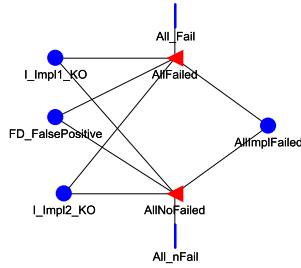


Figure 5.11: Atomic Model of the All Fail Event (*F07_SF_AllFail*)

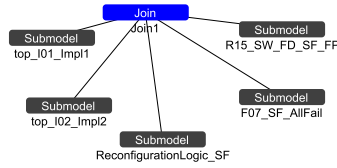


Figure 5.12: Composed Model of the All Fail Event (*top_F07_SF_AllFail*)

Table 5.8: Activities in *F07_SF_AllFail*

Input gate:	AllFailed
Input gate predicate:	$((m(I_Impl1_KO)==1 \quad \&\&$ $m(I_Impl2_KO)==1) \quad $ $m(FD_FalsePositive)==1) \quad \&\&$ $m(AllImplFailed)==0)$
Input gate function:	$m(AllImplFailed)=1;$
Input gate:	AllNoFailed
Input gate predicate:	$((m(I_Impl1_KO)==0 \quad $ $m(I_Impl2_KO)==0) \quad \&\&$ $m(FD_FalsePositive)==0) \quad \&\&$ $m(AllImplFailed)==1)$
Input gate function:	$m(AllImplFailed)=0;$

Figure 5.12 shows the composed model for *all implementations failed* event, linking implementations (*top_I01_Impl1*, *top_I02_Impl2*) - see Figure 5.7 (Impl2 is the same with its corresponding resources), their reconfiguration logic (*ReconfigurationLogic_SF*) - see Figure 5.8, the failure logic (*F07_SF_AllFail*) - see Figure 5.12 and false positive failure event (*R15_SW_SF_SF_FP*) - which is the same as resource models (see Figure 5.3) with its corresponding failure and repair rate.

Reconfiguration subfunction failure

The *reconfiguration subfunction's failure* will occur when (cf. Equation 5.10): (1) all reconfiguration implementations fail (*AllRFailed* event); or (2) reconfiguration implementation's failure is unresolved (*RUnresolved* event).

All reconfiguration implementations failed: *AllRFailed* event occurs when all the reconfiguration implementation fail simultaneously or reconfiguration's fault detection raises a false positive signal. In the model displayed in the Table 5.2, the *AllRFailed* event is defined as follows (cf. Equation 5.11):

$$\begin{aligned}
 \mathcal{F}_{R_1 \text{ O/FP}} &= \mathbf{OR}(\mathcal{F}_{R_Impl1}, \mathcal{F}_{FD_R \text{ FP}}) \\
 \mathcal{F}_{R_2 \text{ O/FP}} &= \mathbf{OR}(\mathcal{F}_{R_Impl2}, \mathcal{F}_{FD_R \text{ FP}}) \\
 \mathcal{F}_{\text{All R Impl.}} &= \mathbf{AND}(\mathcal{F}_{R_1 \text{ O/FP}}, \mathcal{F}_{R_2 \text{ O/FP}})
 \end{aligned}
 \tag{5.15}$$

The event $\mathcal{F}_{\text{All R Impl.}}$ is described in Figure 5.13. The places *I_R_Impl1_KO* and *I_R_Impl2_KO* indicate the state of the reconfiguration subfunction's implementations, *FD_R_FalsePositive* indicates the presence of false positive signals and *AllReconfigFailed* place denotes the event $\mathcal{F}_{\text{All R Impl.}}$. The failure logic is implemented using the input gates *AllR_Failed* and *AllR_Working* (cf. Table 5.9).

Table 5.9: Activities in *F03_R_SF_AllRFailed*

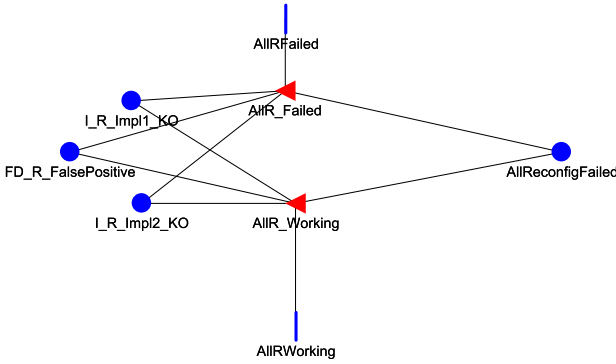


Figure 5.13: Atomic Model of *AllRFailed* Event (*F03_R_SF_AllRFailed*)

Input gate:	AllR_Failed
Input gate predicate:	(((m(I_R_Impl1_KO)==1 && m(I_R_Impl2_KO)==1) m(FD_R_FalsePositive)==1) && m(AllReconfigFailed)==0)
Input gate function:	m(AllReconfigFailed)=1;
Input gate:	AllR_Working
Input gate predicate:	(((m(I_R_Impl1_KO)==0 m(I_R_Impl2_KO)==0) && m(FD_R_FalsePositive)==0) && m(AllReconfigFailed)==1)
Input gate function:	m(AllReconfigFailed)=0;

The logic to reconfigure *R_SF* implementations is in Figure 5.14. The input gate *R_SF_Reconfigure* defines the order of the reconfiguration for the reconfiguration implementations according to their priorities in Table 5.2.

As defined in the *R_SF_Reconfigure* input gate (cf. Table 5.10), the implementation on

Table 5.10: Activities in *ReconfigurationLogic_R_SF*

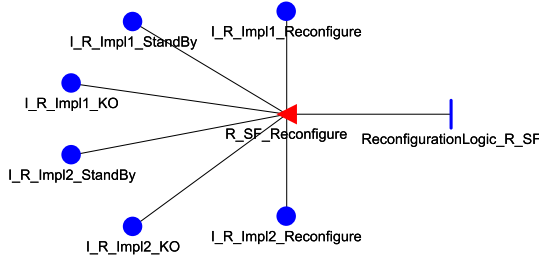


Figure 5.14: Atomic Model of Reconfiguration Implementation's Reconfiguration Logic (*ReconfigurationLogic_R_SF*)

Input gate:	R_SF_Reconfigure
Input gate predicate:	(((m(I_R_Impl1_KO)==1 && m(I_R_Impl2_StandBy)==1) (m(I_R_Impl2_KO)==1 && m(I_R_Impl1_StandBy)==1)))
Input gate function:	if(m(I_R_Impl1_StandBy)==1) m(I_R_Impl1_Reconfigure)=1; else if(m(I_R_Impl2_StandBy)==1) m(I_R_Impl2_Reconfigure)=1;

standby with the highest priority is selected for reconfiguration. Which is implemented according to their position in the "if-else" clauses.

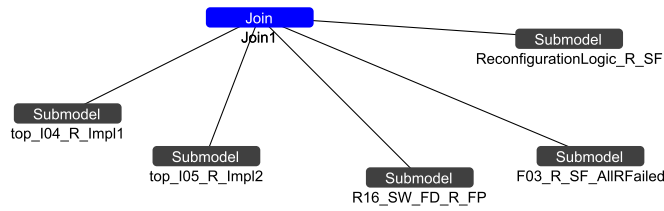


Figure 5.15: Composed Model of the All Reconfiguration Implementation Fail Event (*top_F03_R_SF_AllRFailed*)

The top-level composed model in Figure 5.15 links the following composed models: reconfiguration implementations composed models (*top_I04_R_Impl1*, *top_I05_R_Impl2*); false positive failure model (*R16_SW_FD_R_FP*); the corresponding failure logic model *F03_R_SF_AllRFailed* (cf. Figure 5.13); and reconfiguration logic model *ReconfigurationLogic_R_SF* (cf. Figure 5.14).

Reconfiguration Unresolved Event: the same modelling process applies to the *reconfiguration unresolved* failure event. Applying Equation 5.12 to the Table 5.2:

$$\mathcal{F}_{R_{\text{Unresolved}}} = \text{AND}(\mathcal{F}_{FD_{R_1}}, \mathcal{F}_{FD_{R_2}}) \quad (5.16)$$

Figure 5.16 displays the *reconfiguration unresolved* event model. The operation of the implementations *FD_R_Impl1* and *FD_R_Impl2* together with the input gates *Unresolved* and *NotUnresolved* (cf. table 5.11) determine if the reconfiguration is unresolved.

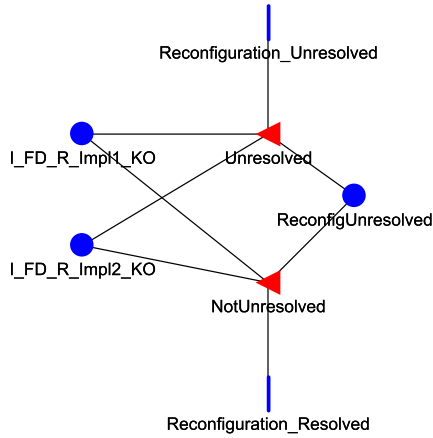


Figure 5.16: Atomic Model of the Reconfiguration Unresolved Event ($F04_R_SF_RUnresolved$)

Table 5.11: Activities in $F04_R_SF_RUnresolved$

Input gate:	Unresolved
Input gate predicate:	$(m(I_FD_R_Impl1_KO)==1 \ \&\& \ m(I_FD_R_Impl2_KO)==1 \ \&\& \ m(ReconfigUnresolved)==0)$
Input gate function:	$m(ReconfigUnresolved)=1;$
Input gate:	NotUnresolved
Input gate predicate:	$((m(I_FD_R_Impl1_KO)==0 \ \ m(I_FD_R_Impl2_KO)==0) \ \&\& \ m(ReconfigUnresolved)==1)$
Input gate function:	$m(ReconfigUnresolved)=0;$

The implementations of the fault detection of the reconfiguration (FD_R_SF) operate in heartbeat/keepalive configuration: all implementations are operating and there are no priorities between them (cf. Figure 5.17, Table 5.12). $FD_R_SF_Reconfigure$ input gate reconfigures all fault detection implementations that go on *standby*.

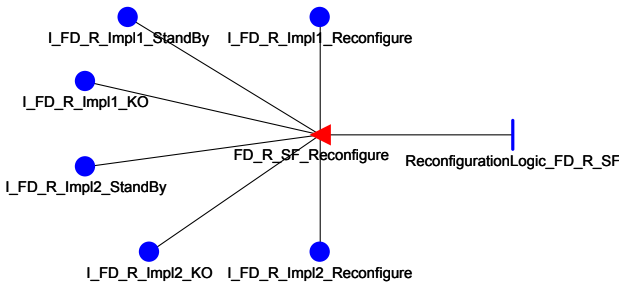


Figure 5.17: Reconfiguration Logic FD_R ($ReconfigurationLogic_FD_R_SF$)

Table 5.12: $ReconfigurationLogic_FD_R_SF$ Activity Characterization

Input gate:	FD_R_SF_Reconfigure
Input gate predicate:	$((m(I_FD_R_Impl1_KO)==1 \ \&\& \ m(I_FD_R_Impl2_StandBy)==1) \ \ (m(I_FD_R_Impl2_KO)==1 \ \&\& \ m(I_FD_R_Impl1_StandBy)==1))$
Input gate function:	$if(m(I_FD_R_Impl1_StandBy)==1) \ m(I_FD_R_Impl1_Reconfigure)=1; \ if(m(I_FD_R_Impl2_StandBy)==1) \ m(I_FD_R_Impl2_Reconfigure)=1;$

The composed model of $\mathcal{F}_{RUnresolved}$ (cf. Figure 5.18) links the following models: composed models of the fault detection of the reconfiguration (FD_R_SF) implementations ($top_I06_FD_R_Impl1$, $top_I07_FD_R_Impl2$); failure unresolved logic model ($F04_R_SF_RUnresolved$); and reconfiguration logic model ($ReconfigurationLogic_FD_R_SF$).

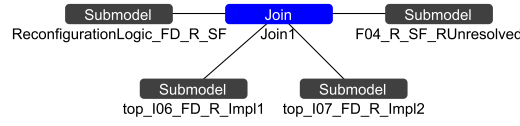


Figure 5.18: Composed Model of Reconfiguration Unresolved Event ($top_F04_R_SF_RUnresolved$)

Reconfiguration Subfunction Failure: Figure 5.19 models the *reconfiguration subfunction failure* event (cf. Equation 5.10). The place SF_R_Failed indicates that the reconfiguration subfunction has failed. This event will be based on the marking of the places $AllReconfigFailed$ (shared with the places in Figure 5.13) and $ReconfigUnresolved$ (shared with the places in Figure 5.16). The behaviour of the model is described by the input gates $SF_Reconfig_Fail$ and $SF_Reconfig_NotFail$ (cf. Table 5.13).

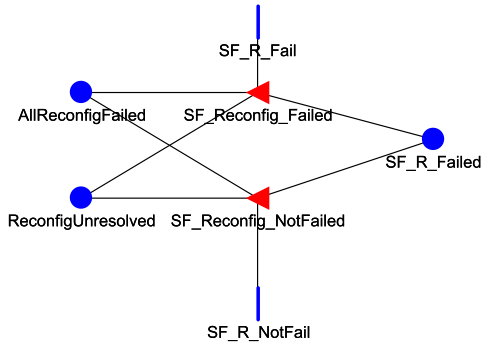


Figure 5.19: Atomic Model of the Reconfiguration SF Fail Event ($F05_R_SF_Failure$)

Table 5.13: Activities in $F05_R_SF_Failure$

Input gate:	$SF_Reconfig_Failed$
Input gate predicate:	$((m(AllReconfigFailed)==1 \quad \quad m(ReconfigUnresolved)==1) \quad \&\& \quad (m(SF_R_Failed)==0))$
Input gate function:	$m(SF_R_Failed)=1;$
Input gate:	$SF_Reconfig_NotFailed$
Input gate predicate:	$(m(AllReconfigFailed)==0 \quad \&\& \quad m(ReconfigUnresolved)==0 \quad \&\& \quad m(SF_R_Failed)==1)$
Input gate function:	$m(SF_R_Failed)=0;$

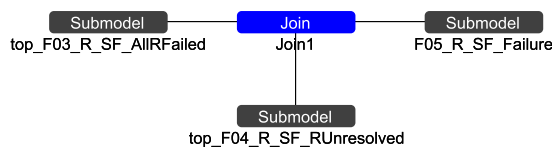


Figure 5.20: Composed Model of the Reconfiguration SF Fail Event ($top_F05_R_SF_Failure$)

The composed model (cf. Figure 5.20) is used to determine the failure probability of the reconfiguration subfunction. Accordingly, it links implementations, resources and their failure logic via shared places: all reconfiguration implementations failed event model: $top_F03_R_SF_AllRFailed$ (cf. Figure 5.13); reconfiguration unresolved event model: $top_F04_R_SF_RUnresolved$ (cf. Figure 5.16); and reconfiguration subfunction failure

logic model: $F_05_R_SF_Failure$ (cf. Figure 5.19).

Fault detection subfunction failure

The *fault detection subfunction failure* is defined in terms of the failure of the fault detection implementation or failure of the function implemented at the destination PU of the monitored function (cf. Equation 5.14). Analysing the implementation #3 in Table 5.2, the fault detection subfunction failure (\mathcal{F}_{FD_SF}) will be determined either by the failure of the fault detection function itself or failure of the control subfunction.

Figures 5.4 and Figure 5.21 show the models of the fault detection and control subfunction implementation failures respectively. In this example neither implementations have redundancies (cf. Table 5.2) and therefore, they are modelled following the same characterization as described in Figure 5.4.

Control subfunction failure: the $I09_ControlSF_NoDependencies$ model (cf. Figure 5.21) indicates that the control subfunction implementation has failed when any of its resources is down (Res1, Res12, Res13). In order to avoid creating logical loops, it is assumed that the failure of the control implementation will be provoked only through its implementations - if dependencies are considered there will be logical loops. Table 5.14 displays the failure logic implemented in the input gates and Figure 5.22 depicts the composed model which links the implementation's failure logic with its constituting resources.

Fault detection subfunction failure: the model which describes the *fault detection subfunction failure* event (cf. Figure 5.23) uses the places $I_FD_SF_KO$ and $I_SF_Control_KO$ from the models $I08_FD_SF_Failure$, $I09_ControlSF_NoDependencies$ respectively.

Table 5.15 describes the logic implemented in the input gates $FD_Failure$ and FD_NoFail and Figure 5.24 shows the composed model of the *fault detection subfunction failure*. The places between the composed models of the control subfunction failure (cf. Figure 5.22), fault detection failure (cf. Figure 5.5) and the fault detection subfunction failure (cf. Figure 5.23) are shared.

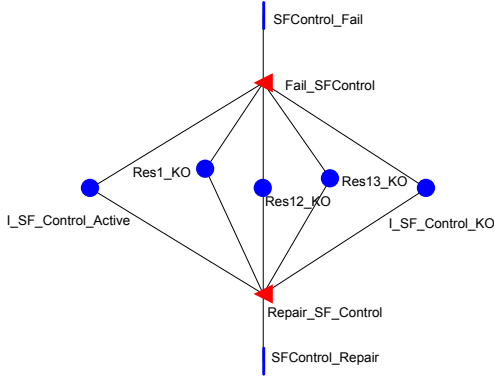


Figure 5.21: Atomic Model of the Control Subfunction Fail Event (*I09_ControlSF_NoDependencies*)

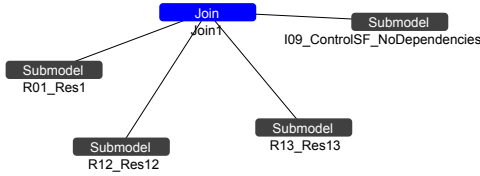


Figure 5.22: Composed Model of the Control Subfunction Fail Event (*top_I09_ControlSF*)

Table 5.14: Activities in *I09_ControlSF_NoDependencies*

Input gate:	Fail_SFControl
Input gate predicate:	$(m(I_SF_Control_KO)==0 \ \&\& \ (m(Res1_KO)==1 \ \&\& \ m(Res12_KO)==1 \ \&\& \ m(Res13_KO)==1))$
Input gate function:	$m(I_SF_Control_Active)=0; \ m(I_SF_Control_KO)=1;$
Input gate:	Repair_SF_Control
Input gate predicate:	$(m(I_SF_Control_KO)==1 \ \&\& \ m(Res1_KO)==0 \ \&\& \ m(Res12_KO)==0 \ \&\& \ m(Res13_KO)==0)$
Input gate function:	$m(I_SF_Control_Active)=1; \ m(I_SF_Control_KO)=0;$

Reconfiguration sequence failure

The *reconfiguration sequence failure* event of the hypothetical system displayed in Table 5.2 is expressed as follows (see also Equation 5.5):

$$\begin{aligned}
 \mathcal{F}_{R_Seq1_SF} &= \mathbf{PAND}(\mathcal{F}_{R_SF}, \mathcal{F}_{SF1} | \text{Active}) \\
 \mathcal{F}_{R_Seq2_SF} &= \mathbf{PAND}(\mathcal{F}_{R_SF}, \mathcal{F}_{SF2} | \text{Active}) \\
 \mathcal{F}_{R_Seq_SF} &= \mathbf{OR}(\mathcal{F}_{R_Seq1_SF}, \mathcal{F}_{R_Seq2_SF})
 \end{aligned} \tag{5.17}$$

Table 5.7 displays nomenclature equivalences between the reconfiguration-related failure equations and the SAN model's events. To create this model previously defined models will be linked using shared places (cf. Figure 5.25): *top_F01_Impl1_FailActive* (cf. Figure 5.9), *top_F05_R_SF_Failure* (cf. Figure 5.20) and PAND gate's model¹⁴ ($\mathbf{PAND}(A,B)$). PAND gate's A and B events are connected with the *SF_R_Failed*

¹⁴A SAN component has been designed which implements the PAND gate's logic for repairable systems. Interested readers please refer to Appendix F to see implementation details and validation.

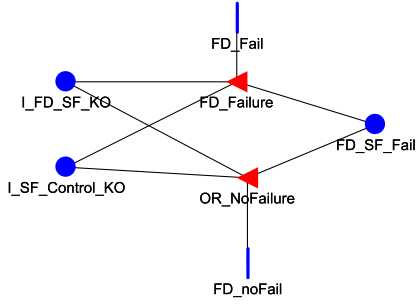


Figure 5.23: Atomic Model of the Fault Detection Subfunction Failure ($F08_FD_SF_Failure$)

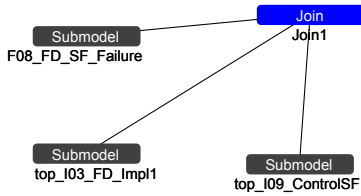


Figure 5.24: Composed Model of the Fault Detection Subfunction Failure ($top_F08_FD_SF_Failure$)

Table 5.15: Activities in $F08_FD_SF_Failure$

Input gate:	FD_Failure
Input gate predicate:	$(m(FD_SF_Fail)==0 \quad \&\&$ $(m(I_FD_SF_KO)==1 \quad \parallel$ $m(I_SF_Control_KO)==1))$
Input gate function:	$m(FD_SF_Fail)=1;$
Input gate:	FD_NoFailure
Input gate predicate:	$(m(FD_SF_Fail)==1 \quad \&\&$ $m(I_FD_SF_KO)==0 \quad \&\&$ $m(I_SF_Control_KO)==0)$
Input gate function:	$m(FD_SF_Fail)=0;$

place of the model $top_F05_R_SF_Failure$ and $FailActive$ place of the model $top_F01_Impl1_FailActive$ respectively (see Equation 5.17). The output place of the PAND gate's model is named $PAND_RF1$.

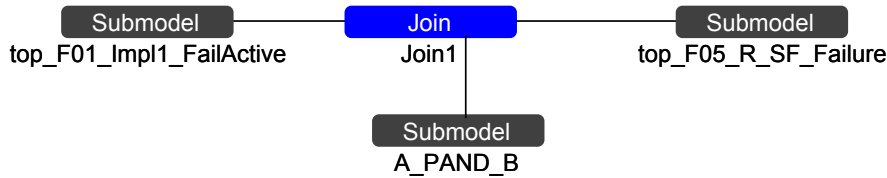


Figure 5.25: Composed Model of the $\mathcal{F}_{R\ Seq,1}$ Event ($top_F06_Impl1_RF1$)

The *reconfiguration sequence failure* event model for the second implementation ($\mathcal{F}_{R\ Seq,2_SF}$) is created following the same logic by connecting: $top_F02_Impl2_FailActive$, $top_F05_R_SF_Failure$ (cf. Figure 5.20) and PAND gate (see Equation 5.17). The output place of the PAND gate's model is named $PAND_RF2$.

Finally, in Figure 5.26, both *reconfiguration sequence failure* models ($\mathcal{F}_{R\ Seq,1_SF}$ and $\mathcal{F}_{R\ Seq,2_SF}$) are linked using the places $PAND_RF1$ and $PAND_RF2$ respectively. The

output of the equation $\mathcal{F}_{R \text{ Seq.}_{_SF}}$ (cf. Equation 5.17) is included in the place RF .

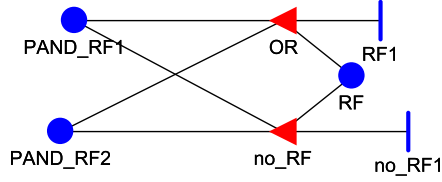


Figure 5.26: Atomic Model of the $\mathcal{F}_{R \text{ Seq.}_{_SF}}$ Event ($F06_SF_RF$)

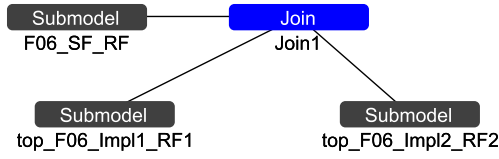


Figure 5.27: Composed Model of the $\mathcal{F}_{R \text{ Seq.}_{_SF}}$ Event ($top_F06_SF_RF$)

Table 5.16: Activities in $F06_SF_RF$

Input gate:	OR
Input gate predicate:	$(m(RF)=0 \&\& (m(PAND_RF1)=1 \vee m(PAND_RF2)=1))$
Input gate function:	$m(RF)=1;$
Input gate:	no_RF
Input gate predicate:	$(m(RF)=1 \&\& m(PAND_RF1)=0 \&\& m(PAND_RF2)=0)$
Input gate function:	$m(RF)=0;$

Figure 5.27 describes the composed model which links *reconfiguration sequence failure* events ($\mathcal{F}_{R \text{ Seq.}_{1_SF}}$, $\mathcal{F}_{R \text{ Seq.}_{2_SF}}$) and the *reconfiguration sequence failure* logic ($\mathcal{F}_{R \text{ Seq.}_{_SF}}$), see Equation 5.17.

Fault detection sequence failure

This development is very similar to the reconfiguration sequence failure model. The *fault detection sequence failure* equations of the configuration displayed in Table 5.2 are expressed as follows (see Equation 5.6):

$$\begin{aligned}
 \mathcal{F}_{FD \text{ Seq.}_{1_SF}} &= \mathbf{PAND}(\mathcal{F}_{FD_SF}, \mathcal{F}_{SF_1} \mid \text{Active}) \\
 \mathcal{F}_{FD \text{ Seq.}_{2_SF}} &= \mathbf{PAND}(\mathcal{F}_{FD_SF}, \mathcal{F}_{SF_2} \mid \text{Active}) \\
 \mathcal{F}_{FD \text{ Seq.}_{_SF}} &= \mathbf{OR}(\mathcal{F}_{FD \text{ Seq.}_{1_SF}}, \mathcal{F}_{FD \text{ Seq.}_{2_SF}})
 \end{aligned} \tag{5.18}$$

Table 5.7 displays nomenclature equivalences between the fault detection related failure equations and the SAN model's events. To characterize the event $\mathcal{F}_{FD \text{ Seq.}_{1_SF}}$ a composed model is created linking the following composed models: $top_F01_Impl1_FailActive$ (cf. Figure 5.9); $top_F08_FD_SF_Failure$ (cf. Figure 5.23); and PAND gate model ($\mathbf{PAND}(A,B)$), where event A is shared with the event

FD_SF_Fail and event B is shared with the event $FailActive$ (see Equation 5.18). The output place of the PAND gate's model is named $PAND_FU1$.

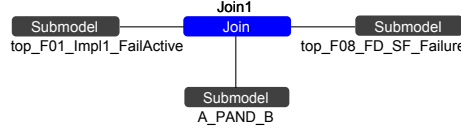


Figure 5.28: Composed Model of the $\mathcal{F}_{FD\ Seq.1}$ Event

Similarly, the event $\mathcal{F}_{FD\ Seq.2_SF}$ is created by linking its corresponding fail active event model ($top_F02_Impl2_FailActive$); fault detection subfunction failure model ($top_F08_FD_SF_Failure$) and PAND gate's failure logic model (see Equation 5.18). The output place of the PAND gate's model is named $PAND_FU2$.

The model in Figure 5.29 links both $\mathcal{F}_{FD\ Seq.1_SF}$ and $\mathcal{F}_{FD\ Seq.2_SF}$ in order to characterize the $\mathcal{F}_{FD\ Seq.}$ event (cf. Equation 5.18). Table 5.17 displays the logic of the input gates $FailUndetected$ and $FailDetected$, which determine the $\mathcal{F}_{FD\ Seq.}$ event in Figure 5.29.

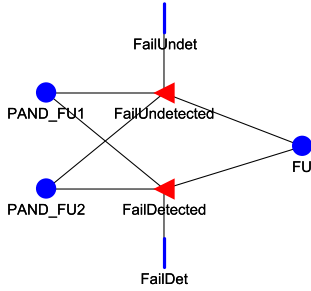


Figure 5.29: Atomic Model of the $\mathcal{F}_{FD\ Seq.}$ Event ($F09_FD_SF_FU$)

Table 5.17: Activities in $F09_FD_SF_FU$

Input gate:	FailUndetected
Input gate predicate:	$(m(FU)==0 \ \&\& \ (m(PAND_FU1)==1 \ \ \ m(PAND_FU2)==1))$
Input gate function:	$m(FU)=1;$
Input gate:	FailDetected
Input gate predicate:	$(m(FU)==1 \ \&\& \ m(PAND_FU1)==0 \ \&\& \ m(PAND_FU2)==0)$
Input gate function:	$m(FU)=0;$

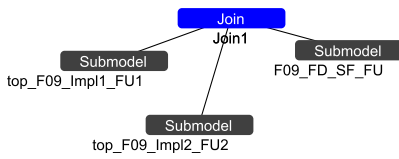


Figure 5.30: Composed Model of the $\mathcal{F}_{FD\ Seq.}$ Event ($top_F09_FD_SF_FU$)

In order to evaluate the failure probability of the $\mathcal{F}_{FD\ Seq.}$ event, the model in Figure 5.30 links composed models $top_F09_Impl1_FU1$ ($\mathcal{F}_{FD\ Seq.1_SF}$) and $top_F09_Impl2_FU2$ ($\mathcal{F}_{FD\ Seq.2_SF}$) with the model $F09_FD_SF_FU$ which determines the event $\mathcal{F}_{FD\ Seq.}$. The places $PAND_FU1$ and $PAND_FU2$ are shared with

the places of the PAND models of the $\mathcal{F}_{\text{FD Seq},1_SF}$ and $\mathcal{F}_{\text{FD Seq},2_SF}$ events respectively. The place FU models the event $\mathcal{F}_{\text{FD Seq},_SF}$.

Input Subfunction Failure

The subfunction failure is defined according to the Equation 5.2. In the case of the input subfunction SF there is no need to consider the influence of dependencies. Therefore, the *input subfunction failure* is determined by the events $\mathcal{F}_{\text{All Impl}}$ and $\mathcal{F}_{\text{Unresolved}}$ where $\mathcal{F}_{\text{Unresolved}} = \mathbf{OR}(\mathcal{F}_{\text{R Seq},_SF}, \mathcal{F}_{\text{FD Seq},_SF})$. Figure 5.31 describes the failure logic of the subfunction SF failure event \mathcal{F}_{SF} .

Table 5.18 displays the failure logic of the input gates $Fail_SF$ and OK_SF . The marking of the FU , RF and $AllImplFailed$ places are determined by linking the model of Figure 5.31 with the previously defined $top_F06_SF_RF$ (cf. Figure 5.27); $top_F09_SF_FU$ (cf. Figure 5.29); and $top_F07_SF_AllFail$ (cf. Figure 5.12) models. Figure 5.32 depicts the composed model which determines the occurrence of the \mathcal{F}_{SF} event.

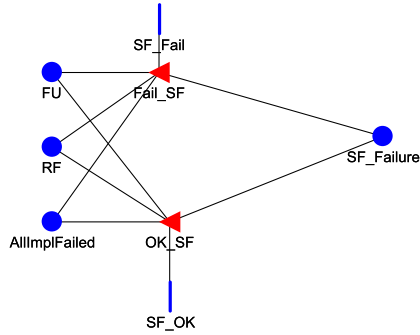


Figure 5.31: Atomic Model of the \mathcal{F}_{SF} Event ($F10_SF_Failure$)

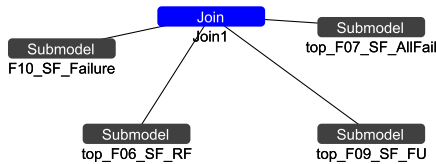


Figure 5.32: Composed Model of the \mathcal{F}_{SF} Event ($top_F10_SF_Failure$)

Table 5.18: Activities in $F10_SF_Failure$

Input gate:	Fail_SF
Input gate predicate:	$((m(FU)==1 \quad \quad m(RF)==1) \quad \&\& \quad m(AllImplFailed)==1) \quad \&\& \quad m(SF_Failure)==0)$
Input gate function:	$m(SF_Failure)=1;$
Input gate:	OK_SF
Input gate predicate:	$(m(FU)==0 \quad \&\& \quad m(RF)==0 \quad \&\& \quad m(AllImplFailed)==0 \quad \&\& \quad m(SF_Failure)==1)$
Input gate function:	$m(SF_Failure)=0;$

Control Subfunction Failure

The *control subfunction failure* is defined according to the Equation 5.2 determined by the events $\mathcal{F}_{\text{All Impl}}$, $\mathcal{F}_{\text{Unresolved}}$ and $\mathcal{F}_{\text{Dependencies}}$. Since the control subfunction has only one implementation (cf. Table 5.2), the failure of the control subfunction will be determined by the failure of the implementation itself or the $\mathcal{F}_{\text{Dependencies}}$ event. There is no $\mathcal{F}_{\text{Unresolved}}$ event because there are no alternate implementations. Figure 5.33 describes the failure logic of the *controlSF* failure event ($\mathcal{F}_{\text{ControlSF}}$). The event is determined by the failure of its resources (Res1, Res12, Res13) or the failure of the input subfunction *SF_Failure* place.

Table 5.19 displays the failure logic of the input gates *Fail_ControlSF* and *OK_ControlSF*. The marking of the *Res1_KO*, *Res12_KO*, *Res13_KO* and *SF_Failure* places are determined by linking the model of Figure 5.33 with the previously defined resources failure/repair models (cf. Figure 5.3) and input subfunction's failure model *top_F10_SF_Failure* (cf. Figure 5.32). Figure 5.34 depicts the composed model which determines the occurrence of the $\mathcal{F}_{\text{ControlSF}}$ event.

Note that we have previously modelled the *control subfunction failure* without input dependencies to avoid creating logical loops with the modelling of the input subfunction's fault detection performance (cf. Figure 5.21). In Figure 5.34 the places of the same resources will be shared accounting for the failure/repair of the resource existing in both models.

Output Subfunction Failure

The *output subfunction failure* is defined according to the Equation 5.2 determined by the events $\mathcal{F}_{\text{All Impl}}$, $\mathcal{F}_{\text{Unresolved}}$ and $\mathcal{F}_{\text{Dependencies}}$. The output subfunction also has only one implementation (cf. Table 5.2) and therefore, the *output subfunction failure* will be determined by the failure of the implementation itself or the $\mathcal{F}_{\text{Dependencies}}$ event. Figure 5.35 describes the failure logic of the *OutputSF* failure event ($\mathcal{F}_{\text{OutputSF}}$) that is determined by the failure of its resources (Res1, Res14, Res15) or the failure of the control subfunction *I_ControlSF_KO* place.

Table 5.20 displays the failure logic of the input gates *Fail_OutputSF* and

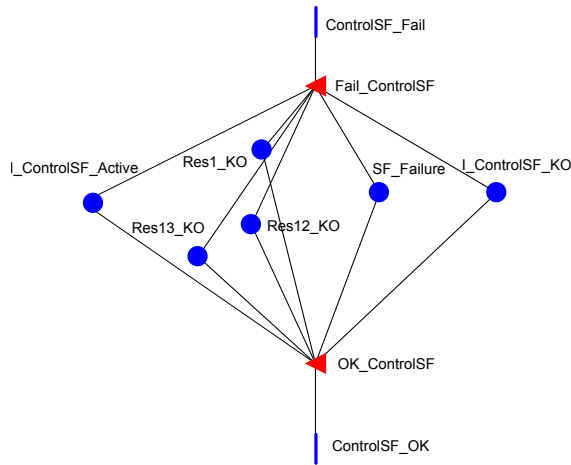


Figure 5.33: Atomic Model of the $\mathcal{F}_{ControlSF}$ Event ($I09_ControlSF$)

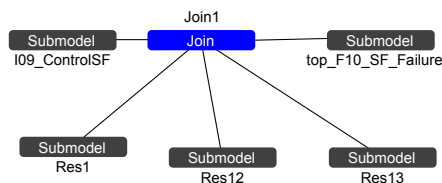


Figure 5.34: Composed Model of the $\mathcal{F}_{ControlSF}$ Event ($top_I09_ControlSF$)

Table 5.19: Activities in $I09_Control_SF_Failure$

Input gate:	Fail_ControlSF	
Input gate predicate:	$((m(Res1_KO)==1$ $m(Res12_KO)==1$ $m(Res13_KO)==1$ $m(SF_Failure)==1)$ $m(I_ControlSF_KO)==0)$	$\&\&$ $\&\&$
Input gate function:	$m(I_ControlSF_Active)=0;$ $m(I_ControlSF_KO)=1;$	
Input gate:	OK_ControlSF	
Input gate predicate:	$(m(Res1_KO)==0$ $m(Res12_KO)==0$ $m(Res13_KO)==0$ $m(SF_Failure)==0$ $m(I_ControlSF_KO)==1)$	$\&\&$ $\&\&$ $\&\&$ $\&\&$
Input gate function:	$m(I_ControlSF_Active)=1;$ $m(I_ControlSF_KO)=0;$	

$OK_OutputSF$. The marking of the $Res1_KO$, $Res14_KO$, $Res15_KO$ and $I_ControlSF_KO$ places are determined by linking the model of Figure 5.35 with the previously defined resources failure/repair models (cf. Figure 5.3) and control subfunction's failure model $top_I09_ControlSF$ (cf. Figure 5.34). Figure 5.36 depicts the composed model which determines the occurrence of the $\mathcal{F}_{OutputSF}$ event and accordingly the failure of the main function.

To evaluate the failure probability of the output subfunction (and the main function), a reward variable is defined characterizing the performance measurement which will indicate the failure of the output subfunction:

```
double reward=0;
if (I10_OutputSF->I_OutputSF_KO->Mark()==1)
    reward+=1;
return(reward);
```

Which in turn is used later to evaluate probabilities.

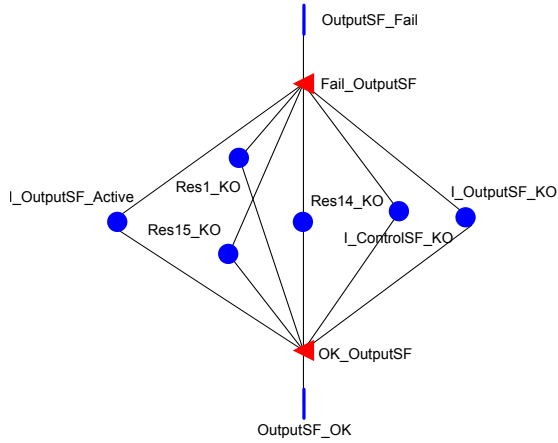


Figure 5.35: Atomic Model of the $\mathcal{F}_{\text{OutputSF}}$ Event ($I10_OutputSF$)

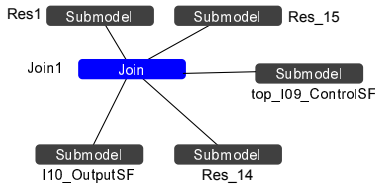


Figure 5.36: Composed Model of the $\mathcal{F}_{\text{OutputSF}}$ Event ($top_I10_OutputSF$)

Table 5.20: Activities in $I10_OutputSF_Failure$

Input gate:	Fail_OutputSF	
Input gate predicate:	$((m(\text{Res1_KO})==1$ $m(\text{Res14_KO})==1$ $m(\text{Res15_KO})==1$ $m(\text{I_ControlSF_KO})==1)$	$\&\&$ $\&\&$
Input gate function:	$m(\text{I_OutputSF_Active})=0;$ $m(\text{I_OutputSF_KO})=1;$	
Input gate:	OK_OutputSF	
Input gate predicate:	$(m(\text{Res1_KO})==0$ $m(\text{Res14_KO})==0$ $m(\text{Res15_KO})==0$ $m(\text{I_ControlSF_KO})==0$ $m(\text{I_OutputSF_KO})==1)$	$\&\&$ $\&\&$ $\&\&$ $\&\&$
Input gate function:	$m(\text{I_OutputSF_Active})=1;$ $m(\text{I_OutputSF_KO})=0;$	

5.4.2 Fire Protection Control

Based on the generic SAN modelling process described in subsection 5.4.1, we have evaluated the unavailability of the Fire Protection Control main function implemented using alternative configurations for the physical location $\text{Train.Car}_1.\text{Zone}_A$. Namely, alternative redundancy and reconfiguration strategies have been tested, as well as the influence of fault detection, reconfiguration and communication implementations on the system failure probability.

The difference from the analysed configurations in Chapter 4 is that the architectures analysed in this chapter are comprised of repairable resources instead of non-repairable resources. Accordingly, we can evaluate the downtime costs taking into account the downtime of the architecture.

Redundancy Strategies

Alternative architecture configurations have been analysed each of them organised with different redundancy strategies (cf. Table 4.4) and same reconfiguration strategy implemented with 2 reconfiguration implementations located in different processing units (Table 5.22 2R distributed configuration). Figure 5.37 displays the relative failure probabilities of these configurations normalized with the architecture without redundancies. All simulations have been carried out with a confidence level of 0.99 and absolute confidence interval of 0.001.

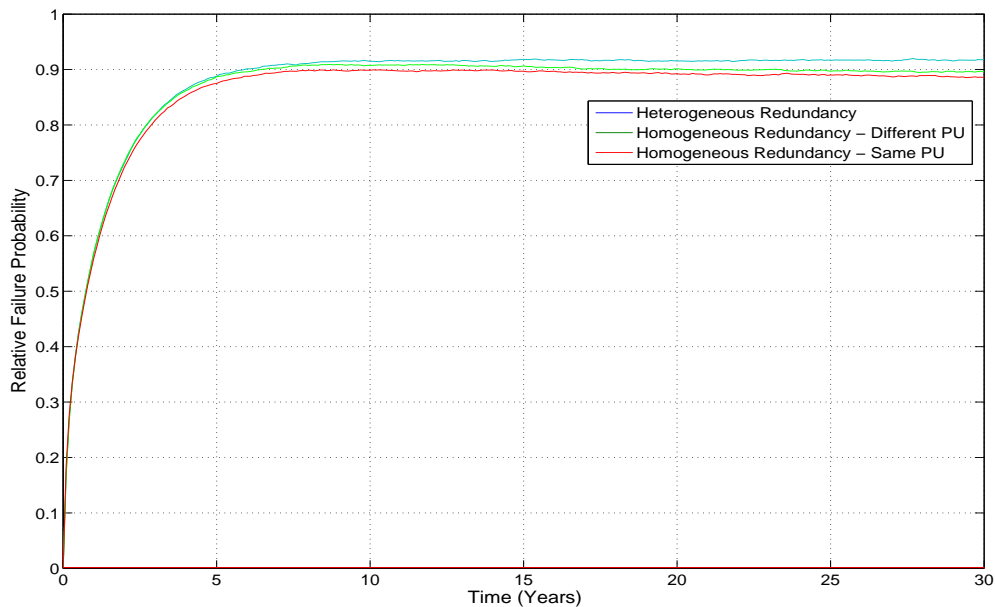


Figure 5.37: Normalized Failure Probability of Fire Protection Control Configurations

Figure 5.37 shows how the use of alternative redundancy strategies improve system's failure probability with respect to the configuration without redundancies. The following improvements have been observed at $T=20$ year time instant with respect to the configuration without redundancies: heterogeneous redundancy 8% better; homogeneous redundancy connected at a different PU 9.4% better; and homogeneous redundancy connected at the same PU 10.2% better.

The configuration with the lowest failure probability is the homogeneous redundancy configuration connected at the same PU (as in Chapter 4). However, with repairable

systems, the failure probability of the heterogeneous redundancy configuration is slightly higher than the homogeneous configurations. This is a consequence of the added extra resources to make implementations compatible (temperature sensor, SW to detect fire, communication). Therefore, we can see that the addition of extra resources worsens the failure probability.

Table 5.21 displays the cost of alternative configurations normalized with respect to the cost of the configuration without redundancies.

Table 5.21: Normalized Cost of Alternative Fire Protection Control Configurations

Configuration	Relative HW/SW/Comm. Cost	Relative Downtime Cost
1 Heterogeneous Redundancy	1.4482	0.89794
1 Homogeneous Redundancy - Same P _{UFP}	1.5322	0.94155
1 Homogeneous Redundancy - Different PU	1.6162	0.94956

Due to the lower hardware/software/communication cost of the heterogeneous redundancy configuration and thanks to the small differences between the failure probabilities of different configurations, the cheapest solution is the heterogeneous redundancy configuration. Note that these result are obtained for the values in Appendix E and assumptions in Section 5.3. Therefore, there may be variations in the results. An analysis of the sensitivity of the cost calculation parameters should be performed here (see Section 5.5).

Reconfiguration Strategies

Table 5.22 displays the influence of alternative reconfiguration strategies on system availability at the time instant T=10 for the heterogeneous redundancy configuration. We carry out different simulations with a confidence level=0.99 and confidence interval=0.0009. In these simulations we consider different failure rate values of health management SW components (λ_{SW_HM}): SW_FD, SW_R and SW_FD_R. The failure rates of these software resources have been modified to highlight the influence of reconfiguration implementations on system unavailability.

From Table 5.22 the following patterns have been identified:

Table 5.22: Fire Protection Control (FPC) Unavailability for Reconfiguration Distribution Strategies (T=10 years)

<i>Configuration</i>	<i>Reconfiguration Implementation Distributions</i>	<i>FPC Unavailability</i>		
		λ_{SW_HM} =0.05	λ_{SW_HM} =0.15	λ_{SW_HM} =0.25
1R Centralised	$\mathbf{PU}_1(\text{R_FireDet}_1)$	0.365	0.366	0.366
2R Centralised	$\mathbf{PU}_1(\text{R_FireDet}_1, \text{R_FireDet}_2)$	0.569	0.569	0.570
2R Distributed	$\mathbf{PU}_1(\text{R_FireDet}_1); \mathbf{PU}_2(\text{R_FireDet}_2)$	0.366	0.366	0.366
3R Centralised	$\mathbf{PU}_1(\text{R_FireDet}_1, \text{R_FireDet}_2, \text{R_FireDet}_3)$	0.568	0.569	0.569
3R Distributed	$\mathbf{PU}_1(\text{R_FireDet}_1); \mathbf{PU}_2(\text{R_FireDet}_2); \mathbf{PU}_3(\text{R_FireDet}_3)$	0.366	0.366	0.366

- The influence of the failure rate of the health management implementations on the main function failure is negligible.
- Centralised configurations perform worse than distributed implementations due to the unique processing unit acting as a common cause failure.
- The number of redundancy implementations within the configurations of the same group (centralised, distributed) does not have an effect on the main function failure. There is really no need of redundancies in this case for reconfigurations.

The failure probability of the fire control algorithm subfunction does not show variations by changing system configurations. However, if we focus on the fire detection subfunction and its underlying failure events there are some characteristics worth mentioning. Table 5.23 shows the failure probability of the fire detection subfunction failure ($\mathcal{F}_{\text{FireDet}}$), fire detection subfunction's reconfiguration sequence failure event ($\mathcal{F}_{\text{R,Seq,FireDet}}$ cf. Equation 5.17), and fire detection subfunction's reconfiguration failure event ($\mathcal{F}_{\text{R_FireDet}}$) for different failure rates of health management implementations (λ_{HM}) and different reconfiguration strategies. All the simulations have been performed with a confidence level = 0.99 and confidence interval = 0.0009.

The performance of the fire detection subfunction shows the influence of the failure rate of the health management implementations and the influence of the distribution of the reconfiguration implementations. Table 5.23 points out the following characteristics:

Table 5.23: Failure Probability of the Fire Detection and its Underlying Events (T=10 years)

Events	$\lambda_{HM} = 0.05$					$\lambda_{HM} = 0.15$					$\lambda_{HM} = 0.25$				
	1R	2RC	2RD	3RC	3RD	1R	2RC	2RD	3RC	3RD	1R	2RC	2RD	3RC	3RD
$\mathcal{F}_{\text{FireDet}}$	0.052	0.355	0.052	0.355	0.05	0.052	0.355	0.052	0.356	0.052	0.052	0.355	0.052	0.356	0.052
$\mathcal{F}_{\text{RSeqFireDet}}$	0.008	0.04	0.005	0.038	0.004	0.011	0.082	0.01	0.073	0.008	0.013	0.115	0.013	0.101	0.011
$\mathcal{F}_{\text{R_FireDet}}$	0.319	0.365	0.276	0.353	0.247	0.578	0.679	0.428	0.623	0.282	0.767	0.925	0.571	0.835	0.391

x-R-Conf: x number of *reconfiguration* implementations in *Conf* configuration, where C = centralised and D = distributed; e.g., 3RD = 3 reconfiguration implementations in distributed configuration

- The failure probability of the centralised configurations is significantly higher than the distributed configurations.
- Increasing the failure rate of the health management implementations with the values shown in Table 5.23 slightly influences the failure probability of the $\mathcal{F}_{\text{FireDet}}$ in 3RC and 3RD configurations.
- Increasing the failure rate of the health management implementations also increases the failure probability of the $\mathcal{F}_{\text{R,Seq,FireDet}}$ and $\mathcal{F}_{\text{R_FireDet}}$ events.
- The greater the number of reconfiguration redundancies, the lower the failure probability of $\mathcal{F}_{\text{R,Seq,FireDet}}$ and $\mathcal{F}_{\text{R_FireDet}}$ events of the same configuration (centralised, distributed) saving the 1R configuration.

Despite an increase in the failure rate of the health management implementations impacts directly on the reconfiguration subfunction failure ($\mathcal{F}_{\text{R_FireDet}}$), its influence on the main function is dependent on a sequence of events (cf. Equation 5.5). Hence, for the event $\mathcal{F}_{\text{FireDet}}$ to happen, $\mathcal{F}_{\text{RSeq_FireDet}}$ must fail according to the sequence dependent constraint. As a result, its influence on the subfunction failure, and accordingly, on the main function failure is attenuated. Therefore, as we can see in Table 5.23, its contribution to the main function failure is not as important as its contribution to the failure of the reconfiguration subfunction itself.

Influence of Health Management Implementations

Taking the heterogeneous redundancy configuration #2 as a starting point (cf. Table 4.4), the influence of the fault detection, reconfiguration and communication implementations have been analysed assuming real and ideal behaviour of each of these implementations.

Figure 5.38 depicts the failure probability values of these configurations in which all the simulations have been performed with confidence level=0.99 and confidence interval=0.001.

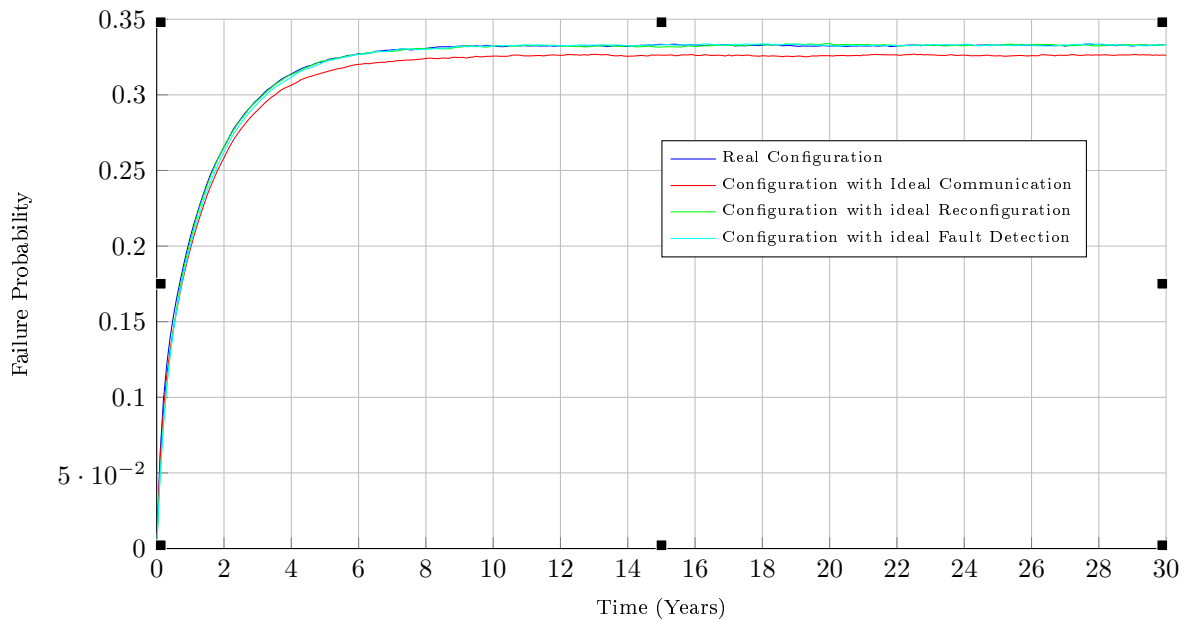


Figure 5.38: Fire Protection Control Failure Probability with Ideal Assumptions

In Figure 5.38 we can see that the influence of the communication is more important than the influence of the reconfiguration and fault detection implementations. For instance at the time instant $T=15$ the following failure probability values hold:

- real configuration = 0.334 ± 0.001 ;
- ideal reconfiguration = 0.333 ± 0.001 ;
- ideal fault detection = 0.331 ± 0.001 ;
- ideal communication = 0.326 ± 0.001 ;

We can see that (1) the influence of the health management implementations is almost negligible and (2) the fault detection implementation has a higher influence than the reconfiguration implementation because the fault detection subfunction does not have redundancies. The impact of the communication implementation on the top-event is even more important because the health management implementations only influence the fire detection input subfunction, whereas communication affects most of the Fire Protection Control main function's subfunctions.

5.4.3 Door Status Control

Based on the generic SAN modelling process described in subsection 5.4.1, we have evaluated the unavailability of the Door Status Control (DSC) main function implemented using alternative configurations for the physical location *Train.Car₁.Zone_A.Door*. Namely, alternative redundancy and reconfiguration strategies have been tested, as well as the influence of fault detection, reconfiguration and communication implementations on the system failure probability.

The difference from the analysed DSC configurations in Chapter 4 is that the architectures analysed in this chapter are comprised of repairable resources instead of non-repairable resources. Accordingly, we can evaluate the downtime costs taking into account the downtime of the architecture.

Redundancy Strategies

Alternative architecture configurations have been analysed, each of them organized with different redundancy strategies using a duplicated reconfiguration implementation located in different processing units for each subfunction with redundancies (Table 5.25 2R centralised configuration). Figure 5.39 depicts the relative failure probabilities of the configurations displayed in the Table 4.7 normalized with the architecture without redundancies #1. All simulations have been carried out with a confidence level of 0.99 and absolute confidence interval of 0.001.

Figure 5.39 shows how the use of alternative redundancy strategies improve system's failure probability with respect to the configuration without redundancies. The follow-

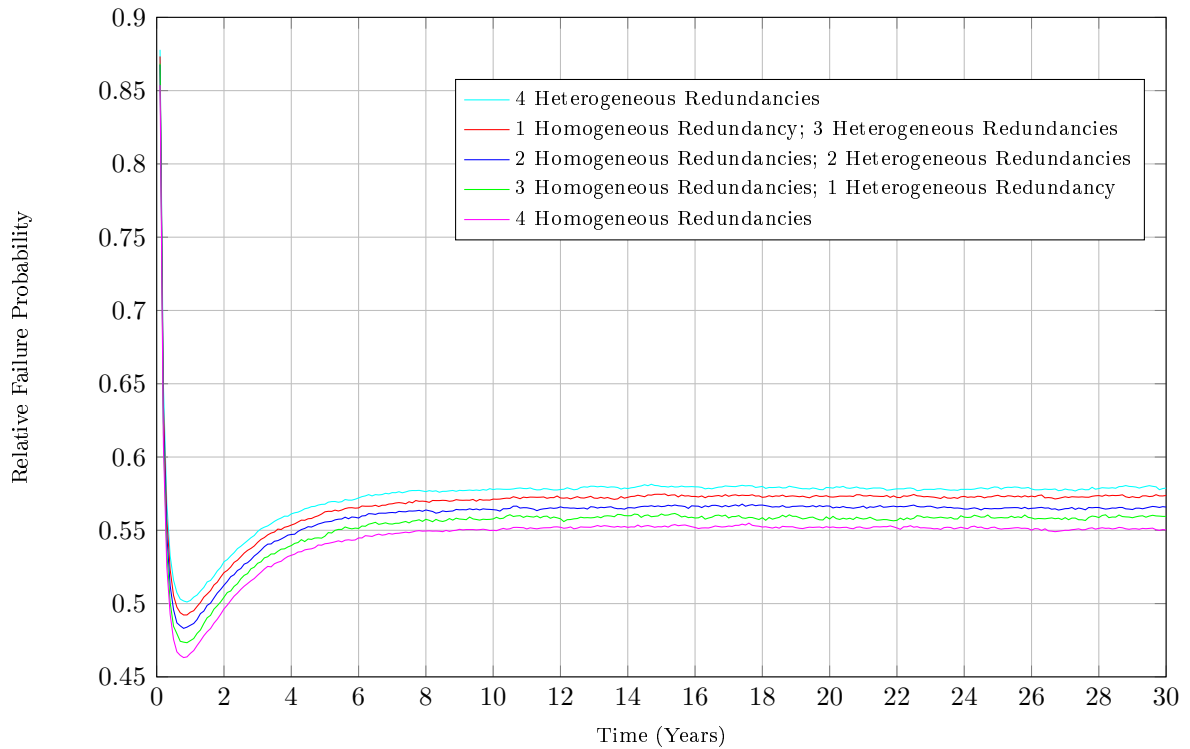


Figure 5.39: Normalized Door Status Control Configurations Failure Probability

ing improvements have been observed at $T=20$ year time instant with respect to the configuration without redundancies: 4 heterogeneous redundancies 42% better; 3 heterogeneous redundancies and 1 homogeneous redundancy 42.57% better; 2 heterogeneous redundancies and 2 homogeneous redundancies 43.23% better; 1 heterogeneous redundancy and 3 homogeneous redundancies 44.07% better; 4 homogeneous redundancies 44.74% better.

Table 5.24 displays the relative costs of alternative configurations normalized with respect to the configuration without redundancies. The cost assessment has been carried out according to the Section 5.3 and using the values shown in Appendix E. When considering the cost of the hardware, software and communication implementations, the cost of the configurations with heterogeneous redundancies is cheaper than homogeneous redundancy configurations. However, when downtime costs are taken into account, the less reliable the architecture, the higher its cost. Therefore, when including downtime costs, the cost of the configurations with heterogeneous redundancies are greater than the configurations with homogeneous redundancies. Compared with the

Fire Protection Control configurations (see Subsection 5.4.2) the failure probability differences between configurations are greater and therefore homogeneous redundancies obtain the better reduction in cost.

Table 5.24: Normalized Cost of Alternative Door Status Control Configurations

Configuration	Relative HW/SW/Comm Cost	Relative Downtime Cost
4 Heterogeneous Redundancies	1.2212	0.58301
3 Heterogeneous Redundancies; 1 Homogeneous Redundancy	1.2488	0.57689
2 Heterogeneous Redundancies; 2 Homogeneous Redundancies	1.2811	0.57097
1 Heterogeneous Redundancy; 3 Homogeneous Redundancies	1.3088	0.56237
4 Homogeneous Redundancies	1.2903	0.55697

Reconfiguration Strategies

Table 5.25 displays the influence of alternative reconfiguration strategies on system availability at the time instant $T=10$ using a heterogeneous redundancy for each subfunction. We carry out different simulations with a confidence level=0.99 and confidence interval=0.0009. In these simulations we consider different failure rate values of health management SW components (λ_{SW_HM}): SW_FD, SW_R and SW_FD_R. The failure rates of these software resources have been modified to highlight the influence of reconfiguration implementations on system unavailability.

Table 5.25 displays that the distribution of the selected reconfiguration implementations for the Door Status Control main function does not have any influence on the final system's failure probability. However, note that these results cannot be compared with the values displayed in Table 5.22. This is because there is only one subfunction with redundancies in the Fire Protection Control case and we named centralised reconfigurations those strategies which centralize redundant reconfiguration implementations in the same PU. In the Door Status Control case centralised reconfigurations group all the subfunction's redundancies with the same priority in the same PU.

Table 5.25: Door Status Control (DSC) Unavailability for Reconfiguration Distribution Strategies (T=10 years)

<i>Configuration</i>	<i>Reconfiguration Implementation Distributions</i>	<i>DSC Unavailability</i>		
		$\lambda_{SW_HM}=0.05$	$\lambda_{SW_HM}=0.15$	$\lambda_{SW_HM}=0.25$
1R Centralised	$\mathbf{PU}_1(\mathbf{R_DOD}_1, \mathbf{R_DCD}_1, \mathbf{R_OD}_1, \mathbf{R_DV}_1)$	0.346	0.347	0.347
1R Distributed	$\mathbf{PU}_1(\mathbf{R_DOD}_1); \mathbf{PU}_2(\mathbf{R_DCD}_1); \mathbf{PU}_3(\mathbf{R_OD}_1); \mathbf{PU}_4(\mathbf{R_DV}_1)$	0.347	0.347	0.347
2R Centralised	$\mathbf{PU}_1(\mathbf{R_DOD}_1, \mathbf{R_DCD}_1, \mathbf{R_OD}_1, \mathbf{R_DV}_1); \mathbf{PU}_2(\mathbf{R_DOD}_2, \mathbf{R_DCD}_2, \mathbf{R_OD}_2, \mathbf{R_DV}_2)$	0.347	0.347	0.347
2R Distributed	$\mathbf{PU}_1(\mathbf{R_DOD}_1, \mathbf{R_DCD}_2); \mathbf{PU}_2(\mathbf{R_DOD}_2, \mathbf{R_DCD}_1); \mathbf{PU}_3(\mathbf{R_OD}_1, \mathbf{R_DV}_2); \mathbf{PU}_4(\mathbf{R_OD}_2, \mathbf{R_DV}_1)$	0.347	0.347	0.347
3R Centralised	$\mathbf{PU}_1(\mathbf{R_DOD}_1, \mathbf{R_DCD}_1, \mathbf{R_OD}_1, \mathbf{R_DV}_1); \mathbf{PU}_2(\mathbf{R_DOD}_2, \mathbf{R_DCD}_2, \mathbf{R_OD}_2, \mathbf{R_DV}_2); \mathbf{PU}_3(\mathbf{R_DOD}_3, \mathbf{R_DCD}_3, \mathbf{R_OD}_3, \mathbf{R_DV}_3)$	0.347	0.347	0.347
3R Distributed	$\mathbf{PU}_1(\mathbf{R_DOD}_1, \mathbf{R_DCD}_2, \mathbf{R_OD}_3); \mathbf{PU}_2(\mathbf{R_DOD}_2, \mathbf{R_DCD}_1, \mathbf{R_DV}_3); \mathbf{PU}_3(\mathbf{R_DOD}_3, \mathbf{R_OD}_1, \mathbf{R_DV}_2); \mathbf{PU}_4(\mathbf{R_DCD}_3, \mathbf{R_OD}_2, \mathbf{R_DV}_1)$	0.347	0.347	0.347

The failure probability of the door control algorithm subfunction does not show variations changing system configurations. However, if we focus on the input subfunctions and their underlying failure events there are some characteristics worth mentioning. Table 5.26 shows the failure probability of the door closed detection failure event (\mathcal{F}_{DCD}), its corresponding reconfiguration sequence failure event ($\mathcal{F}_{R_Seq_DCD}$), and the reconfiguration subfunction failure event (\mathcal{F}_{R_DCD}). These events have been analysed for different configurations and alternative values of the health management implementation's failure rates. We do not have included the remainder of input subfunctions (door open detection, obstacle detection and door velocity) and their corresponding failure events because all the input subfunctions are characterized equally (i.e., same number and distribution of redundancy, reconfiguration and fault detection implementations). Besides note that we do not have included the 1R distributed configuration for simplification (see Table 5.25 for the configurations).

From Table 5.26 the following characteristics have been identified:

- As the number of reconfiguration's redundancy implementations increase, the fail-

Table 5.26: Failure Probability of the Underlying Events of the Door Status Control Main Function (T=10 years)

Events	$\lambda_{HM} = 0.05$					$\lambda_{HM} = 0.15$					$\lambda_{HM} = 0.25$				
	1RC	2RD	2RC	3RD	3RC	1RC	2RD	2RC	3RD	3RC	1RC	2RD	2RC	3RD	3RC
\mathcal{F}_{DCD}	0.043	0.043	0.043	0.043	0.043	0.043	0.043	0.043	0.043	0.043	0.043	0.043	0.043	0.043	0.043
$\mathcal{F}_{R.Seq_DCD}$	0.013	0.005	0.005	0.003	0.004	0.014	0.008	0.009	0.007	0.007	0.016	0.011	0.011	0.009	0.009
\mathcal{F}_{R_DCD}	0.312	0.138	0.140	0.124	0.127	0.571	0.313	0.316	0.274	0.275	0.761	0.466	0.466	0.390	0.391

ure probability of the reconfiguration sequence failure event ($\mathcal{F}_{R.Seq_SF}$) as well as the reconfiguration subfunction's failure probability (\mathcal{F}_{R_SF}) decreases.

- Despite the effect on the failure probability of the reconfiguration subfunction failure (\mathcal{F}_{R_SF}) is significant for all the configurations, when this event is combined with other events ($\mathcal{F}_{R.Seq_SF}$) the difference between alternative configurations becomes lower due to the sequence dependent constraint (see Equation 5.5).
- As the failure rate of the health management implementations increases, the failure probability of reconfiguration sequence ($\mathcal{F}_{R.Seq_SF}$) and reconfiguration subfunction failure events (\mathcal{F}_{R_SF}) also increase.
- The failure probability of the subfunction failures (\mathcal{F}_{SF}) are not influenced neither by the number of redundancies nor increased failure rate of health management implementations, i.e., the conclusions from Table 5.25 are also seen here.

Influence of Health Management Implementations

Taking the configuration with 4 heterogeneous redundancies as a starting point (cf. Table 4.7, configuration #2), the influence of the fault detection, reconfiguration and communication implementations have been analysed assuming ideal and real behaviour of each of these implementations.

Figure 5.40 displays the failure probability values of these configurations with the confidence level=0.99 and the confidence interval=0.001.

In Figure 5.40 the influence of the communication is more important than

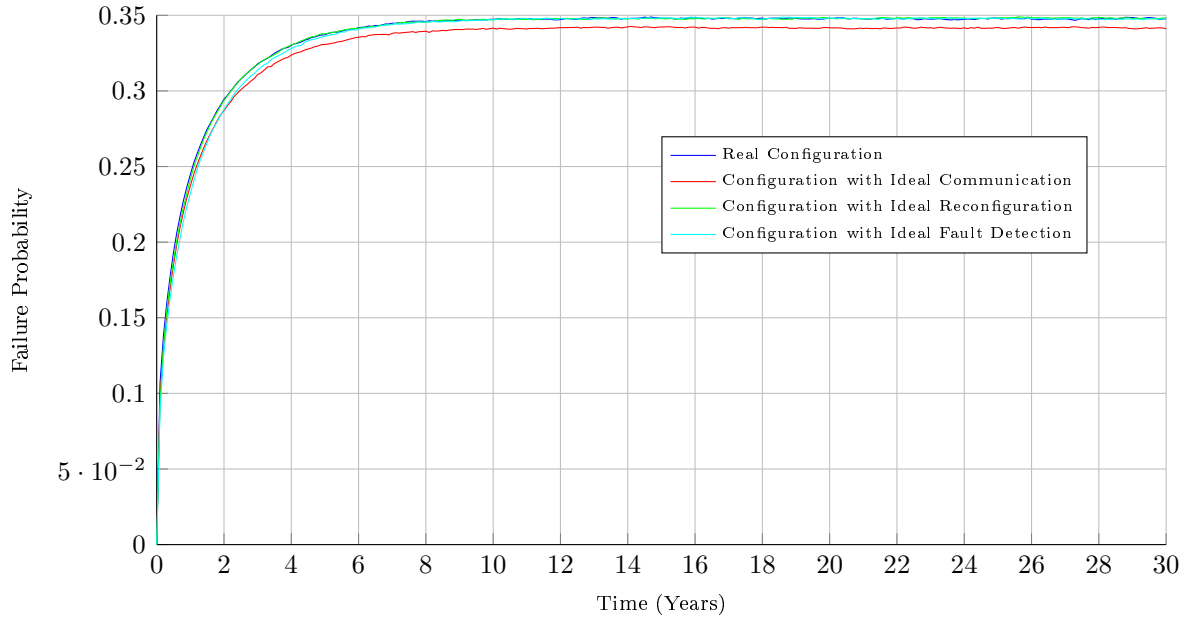


Figure 5.40: Door Status Control Failure Probability with Ideal Assumptions

health management implementations. Again this is because the communication influences many subfunctions and implementations at the same time and health management implementations do not. For instance, at $T=15$ the following failure probability values hold:

1. real configuration = 0.348 ± 0.001 ;
2. ideal communication = 0.342 ± 0.001 ;
3. ideal reconfiguration = 0.347 ± 0.001 ;
4. ideal fault detection = 0.347 ± 0.001 ;

In this case, there is no difference in the influence of fault detection and reconfiguration implementations and their influence can be considered negligible.

5.5 Conclusions

Throughout this chapter we have assumed that the repair process of resources starts as soon as a resource fails and we also have assumed that the repaired resource is as good

as a new one. In this direction there may be some points worth analysing:

- It may be possible to implement preventive maintenance strategies so that a component is repaired/replaced before its failure.
- The degradation of the resource after reparation can be considered by worsening the failure rate after each reparation.
- The influence of alternative SW implementations have been analysed by changing their failure rates. One can also evaluate the influence of the repair rates of elements on system's failure probability to optimize repair and maintenance parameters.

Depending on the design-specific decisions for each main function, the influence on dependability and cost varies. As confirmed in this chapter (and in Chapter 4), optimisation of design decisions with respect to the type and number of redundancy and reconfiguration strategies are feasible to maximize dependability and minimize the cost.

The influence of redundancies on system dependability and cost depend on the analysed main function and its configuration. There are different factors that influence dependability and cost. Concerning the dependability:

- Number of redundancy implementations: the greater the number of redundancies, the lower the failure probability of the subfunction.
- Type of redundancy implementations: generally speaking the failure probability of the heterogeneous redundancies is higher than homogeneous redundancies due to the added extra resources in order to make implementations compatible (e.g., SW implementations, communication).
- Number of reconfiguration's redundancies: while it contributes directly to the improvement of the reconfiguration subfunction's performance, its effect on the main function is attenuated by intermediate sequence of events and it is usually negligible.
- Type of redundancy strategy: distributed reconfiguration redundancies have shown a lower failure probability of the reconfiguration subfunction than the centralised reconfiguration redundancies which concentrate in a single processing unit all the redundancy implementations (see Table 5.22);

Depending on the configuration of the main function, the influence of the type and number of redundancy and reconfiguration mechanism varies. When a subfunction does not have redundancies (i.e., it is a single point of failure), its contribution to the main function failure is more important than influence of the alternative design decisions for another subfunction of the main function that has redundancies (type of redundancies; number and type of reconfiguration strategies).

As for the cost influences:

- Downtime cost: the influence of the downtime cost is higher with less reliable architectures and it is more penalising than the cost incurred by HW, SW or communication resources.
- Type of redundancies: the architecture cost (HW, SW and communications resource cost) of heterogeneous redundancies are cheaper than homogeneous redundancies, however, when downtime costs are included the cost depends on the system's unavailability (which is better for homogeneous redundancies);
- Type of heterogeneous redundancies: heterogeneous redundancies arising from natural compatibility does not need a specific software, whereas heterogeneous redundancies arising from forced compatibility requires fit-for-purpose software which increases its cost.
- Number of heterogeneous redundancies: if there are similar heterogeneous redundancies arising from forced compatibilities, the cost of each heterogeneous redundancy is lower. This happens because the software development cost of one software resource (which is assumed to be valid for all redundancies with slight modifications) is divided among the similar heterogeneous redundancies. Therefore, the cost per each heterogeneous redundancy is not as high as for a single (independent) heterogeneous redundancy.

The sensitivity of the DEM approach to the cost calculation parameters should also be addressed to obtain a higher degree of confidence in the obtained results.

For the analysed configurations in Subsection 5.4.2 and Subsection 5.4.3, the following conclusions are extracted: the influence of the communication implementations on system dependability is not negligible and cannot be considered ideal (see Figure 5.38, and Figure 5.40). Depending on the number of input, control or output subfunc-

tion's redundancies, the influence of health management implementations gains significance. In the developed examples the failure of the fault detection is more influential because it does not have redundancies. The higher the number of redundancies of health management implementations, the lower their failure probability and higher the system cost, but their failure probability improvement is very small (see Table 5.23 and Table 5.26).

The DEM approach would benefit from the automatic extraction of the dependability evaluation model so that the designer is not exposed to error-prone tasks. Besides, the automation would allow us to implement optimization algorithms so that it is possible to explore the design space with alternative architecture configurations (with variations in the number and type of reconfiguration and redundancy strategies) and choose the best architecture according to the given dependability and cost requirements.

The time needed to carry out the simulation of the dependability evaluation model is considerable. This issue originates from the level of (detail and) complexity of the dependability evaluation model and the required accuracy of the results.

D3H2 Methodology: Experimental Evaluation

To proof the feasibility of the D3H2 methodology in real applications, a key application concept in our methodology has been validated: we have added reconfiguration capabilities to existing hardware train network components to recover the system from failures at runtime using heterogeneous redundancies.

In this chapter we present details about the performed experiments [[Aizpurua14](#)]. The chapter is organised as follows:

- Section 6.1 introduces the motivation of this chapter.
- Section 6.2 overviews current industrial railway communication architectures and devices.
- Section 6.3 describes the developed application scenarios in order to validate the concepts treated throughout this dissertation.
- Section 6.4 sets the conclusion of this chapter.

6.1 Introduction

The architecture of the train communication systems is designed with respect to the system functions and their criticality. The data is transmitted from different communication networks according to the criticality of the function.

The reuse of resources emerged from over-dimensioned design decisions is a challenge in the railway domain. Namely, when designing safety-critical functions, the reuse of

resources may pose hazardous consequences that prevent the system from using heterogeneous redundancies. However, the reuse of elements with information, entertainment, or comfort related functions is feasible because they do not pose hazardous consequences, e.g., the failure of Air Conditioning Control or Light Control main functions.

According to the D3H2 methodology, the reuse of resources requires modifications in the system HW/SW architecture, namely: (1) fit the system's PUs with fault detection and reconfiguration mechanisms; (2) design the system with a communication protocol which enables the runtime addition or removal of communication channels; and (3) allocate the reconfiguration table to the reconfiguration decision PU(s) which will indicate the implementation to be reconfigured.

6.2 Industrial Railway Communication Architectures

In Subsection 6.2.1 we describe the main communication networks and in Subsection 6.2.2 we present the communication/processing devices which constitute the train communication architecture.

6.2.1 Communication Networks

Trains have a standard form of data communication specified in the Train Communication Network (TCN) standard IEC 61375 [IEC07]. TCN is a real-time data network comprised of an architecture inter-connecting train vehicles and equipments within a vehicle. The TCN standard specifies Wire Train Bus (WTB) for the inter-connection of vehicles and Multi-function Vehicle Bus (MVB) for intra-vehicle device communication (cf. Figure 6.1). In this work we focus on the communication within a vehicle using MVB.

MVB operates in master-slave configuration of the devices in a vehicle. In this evaluation, the following types of devices are considered: intelligent devices participating in the message communication with administration capabilities or connected I/O elements. The master guarantees deterministic medium access managing periodic and sporadic access to the bus. The communication in MVB follows the publisher/subscriber paradigm:

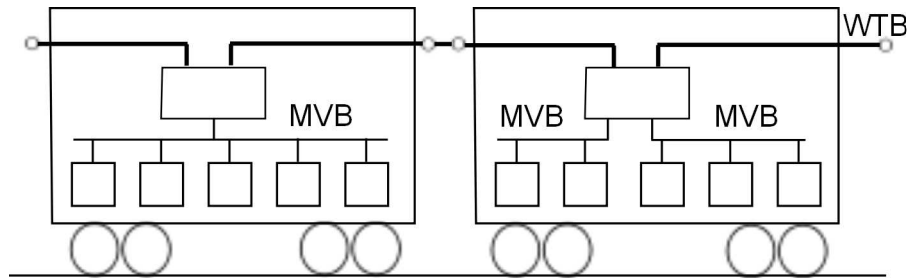


Figure 6.1: TCN Configuration Example [IEC07]

a publisher broadcasts variables and this information is distributed to the subscribers. To this end, a traffic store is implemented; each device holds the variables it produces/-consumes in a shared memory that is a partial copy of the whole network's distributed database.

6.2.2 Communication Devices

All the used devices have been designed and produced by CAF Power & Automation to operate on trains, meeting the rail standards in effect. Thanks to the modularity and flexibility of each module, the needs of each specific application can be achieved by changing the settings of the modules.

The explanation of the characteristics of the devices are limited because they are part of the Intellectual Property of CAF Power & Automation. All devices provide total immunity to electromagnetic interferences in compliance with the standard EN50121-3-2: Railway Applications - Electromagnetic Compatibility.

Every system developed in CAF Power & Automation has three basic functions:

1. Control of communications between the equipment of a train car: the system provides a TCN communication channel for all the train equipment and controls all the information transmitted at the vehicle bus level.
2. Interface with the train via its I/O channels and execution of the train logic: the system is equipped with RS485 series digital and communication modules arranged along the whole train to diagnose and check the train status and to operate in accordance.

- Supervision, monitoring and recording of the train performance: from the driver terminal various train settings and parameters can be entered as well as the condition of every train system can be seen.

We focus on two generic devices to construct the HW/SW architecture so as to test some concepts treated in this dissertation: (1) Communication Interface Card (TICO) and (2) Ethernet communication switch.

The *Communication Interface Card (TICO)* board (cf. Figure 6.2) has a CPU and a FPGA separating communication controllers and application/control/supervision tasks. It has uClinux operating system and its RTAI real-time extension. Therefore, it is possible to combine both real-time and non real-time tasks.

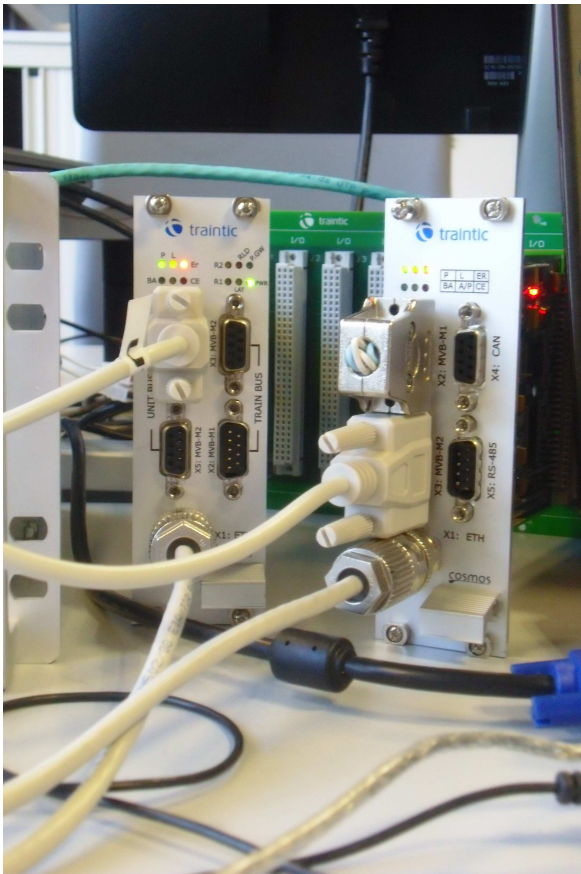


Figure 6.2: *TICO* Board

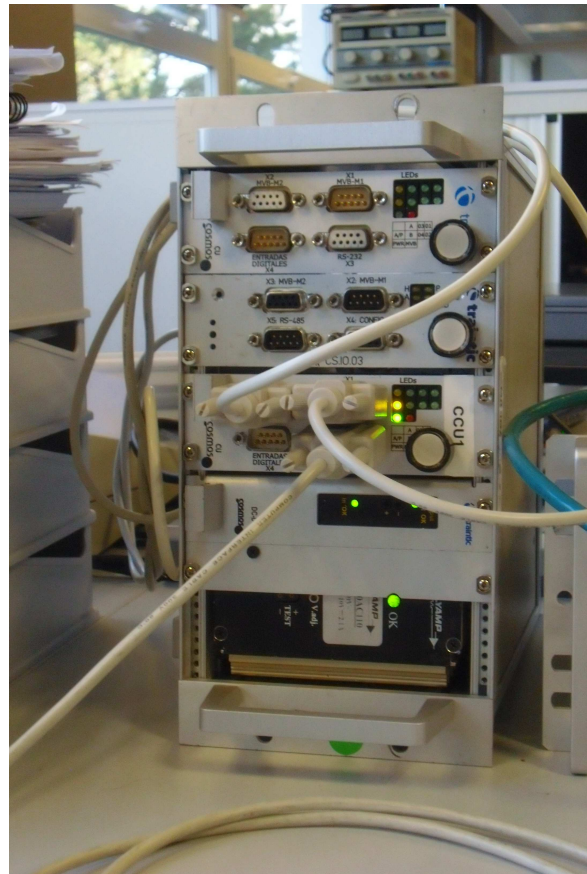


Figure 6.3: *CCU/BA* Module

The *TICO* generic board is expanded into the following application-specific modules: *Multi Interface board Module (MIM)* and *Control and Communication Unit - Bus Administrator (CCU/BA)* module.

The main function of the *CCU/BA* module (cf. Figure 6.3) is to carry out the logic of the control of the train and to administrate the MVB bus (master) in the train car for which it has been devised (BA). The CCU part decides which signals must be used and the BA part controls all information exchange among all the equipment connected to the MVB. Each *CCU/BA* device periodically executes the information transmission commands via MVB and it is able to communicate through MVB and RS485 physical buses.

There shall always be a single *CCU/BA* with active control condition, while other *CCU/BA* devices shall be in passive control condition waiting for an intervention request (on standby).

MIM module (cf. Figure 6.4): it is constituted by a *TICO* board in conjunction with other I/O boards integrated within a backplane. It provides multiple I/O interfaces and control/processing capability with very low power consumption. It is able to communicate through Ethernet, Controller Area Network (CAN), MVB and RS485 physical buses and it can implement any user application with supervisory or control logic.



Figure 6.4: *MIM* Module

The *MIM* module contains different numbers of digital and analogue I/O signal channels.

Ethernet communication switch (cf. Figure 6.5): it permits the communication of different devices connected through an Ethernet communication bus.

The buses and variable movement between buses is supported by the proprietary CStools tool (cf. Figure 6.6). This tool creates the software framework according to the designed communication buses and the user has to add the logic inside the framework (according to the access functions). Uploading the application into the board is

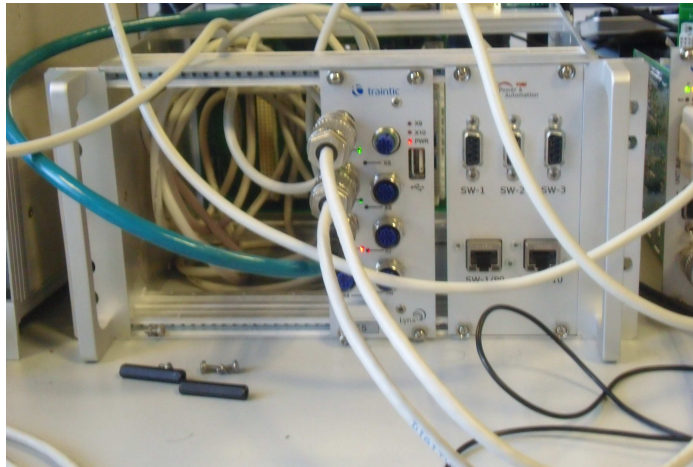


Figure 6.5: Ethernet Switch

done through an Ethernet connection and any File Transfer Protocol (FTP) client.

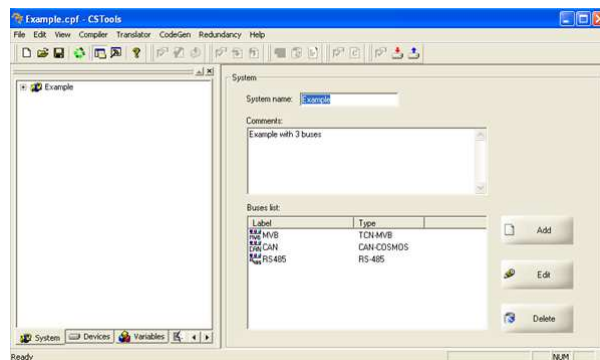


Figure 6.6: Snapshot of the CSTools Configuration Software

Basic Configuration: Figure 6.7 shows the basic configuration from which alternative scenarios have been created to test alternative scenarios (as explained in Section 6.3). Figure 6.8 presents the schematic configuration of the Figure 6.7.

Two *TICO* boards ($TICO_1$, $TICO_2$) are connected to both Ethernet and MVB communication networks. The *CCU/BA* board manages the communication through the MVB network and the Ethernet switch enables the communication of the *TICO* boards through Ethernet. Furthermore, a laptop is used for diagnostic purposes so that it manages the data that flows through the *CCU/BA* module (MVB) and the data that flows through Ethernet.

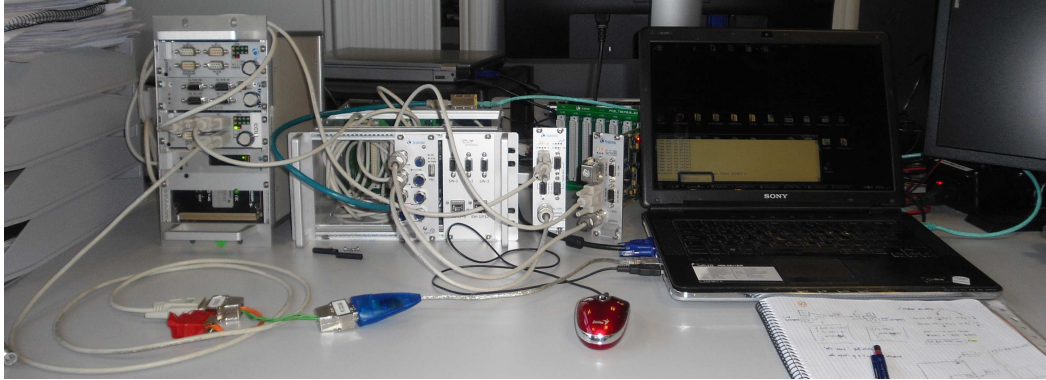


Figure 6.7: Tested Real Configuration

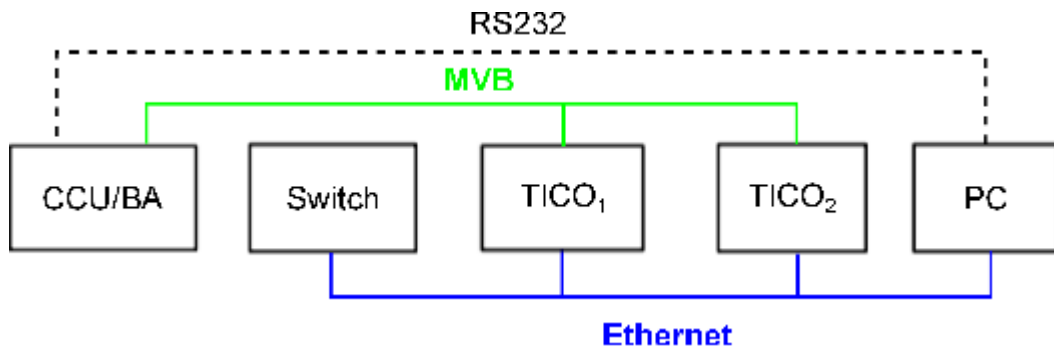


Figure 6.8: Schematic Configuration of the Figure 6.7

6.3 Application Architecture

For the implementation of the reconfiguration process, we identify two phases:

1. Construction of the reconfiguration table: design-time or run-time determined reconfiguration strategies;
2. Activation/deactivation of configurations: reconfiguration techniques.

Run-time construction of the reconfiguration table allows higher flexibility, but requires exploring the architecture dynamically. For safety and predictability purposes, design-time determined reconfiguration strategies are adopted in this study. Regarding the activation or deactivation of configurations, while reconfiguration channels fixed at design-time reduce design complexity, reconfiguration channels established at run-time reduce processing cost and bandwidth by creating redundant communication channels exclusively when their need arises.

In a train there are safety-critical functions which must meet hard real-time constraints (e.g., Door Status Control) and these functions are transmitted through MVB. Besides, other communication protocols coexist in the train; for instance, Ethernet communication protocol transports non-critical information, entertainment or comfort related data. Ethernet provides more flexibility to perform architectural modifications at runtime at the expenses of losing predictability with respect to MVB. There exist other communication networks in a train (e.g., CAN), but this proof of concept has been focused on MVB and Ethernet.

Therefore, the following design decisions have been adopted: MVB has been used for reconfiguration channels fixed at design-time and Ethernet for reconfiguration channels established at run-time. On one hand, communication channels using MVB are obtained by assigning reconfiguration routes at design-time and activating them from the outset. The bandwidth consumption of these redundant communications is constant but their processing is activated solely when their need rise up. On the other hand, in Ethernet, run-time modifications are effectuated using UDP communication threads in client/server like configurations. Communication threads are created and deleted as their need arises, so that the bandwidth and processing needs change exclusively in case of reconfiguration.

The following reconfiguration scenarios (SC) have been tested:

- SC1: sensor reconfiguration: communication route changes to handle sensor failures using heterogeneous redundancies.
- SC2: communication reconfiguration: switching the communication protocol to handle communication failures using heterogeneous redundancies.
- SC3: processing unit reconfiguration: replacing the processing unit and communication routes to handle processing unit failures using homogeneous redundancies.

Three reconfiguration attributes define the reconfiguration space of these scenarios:

- Reconfiguration *granularity* comprehends task or node level reconfigurations. Task level reconfiguration is performed by changing a single task, and node level reconfiguration is performed by changing the whole node (PU and corresponding tasks).

- Reconfiguration *object* addresses SW, HW and communication (Comm.) level reconfigurations: SW reconfigurations modify the SW implementation changing its parameters or structure; HW reconfigurations involve changing the complete HW device; and communication reconfiguration modifies nominal communication routes with alternative ones.
- Reconfiguration channel *activation time* comprehends design-time or run-time activation of communication channels. Note that design-time activation of communication channels really activate at system start-up.

Figure 6.9 describes the reconfiguration space of the tested scenarios. For instance, all the scenarios perform task-level and communication-level reconfigurations, but only SC3 addresses node-level and communication-level reconfigurations.

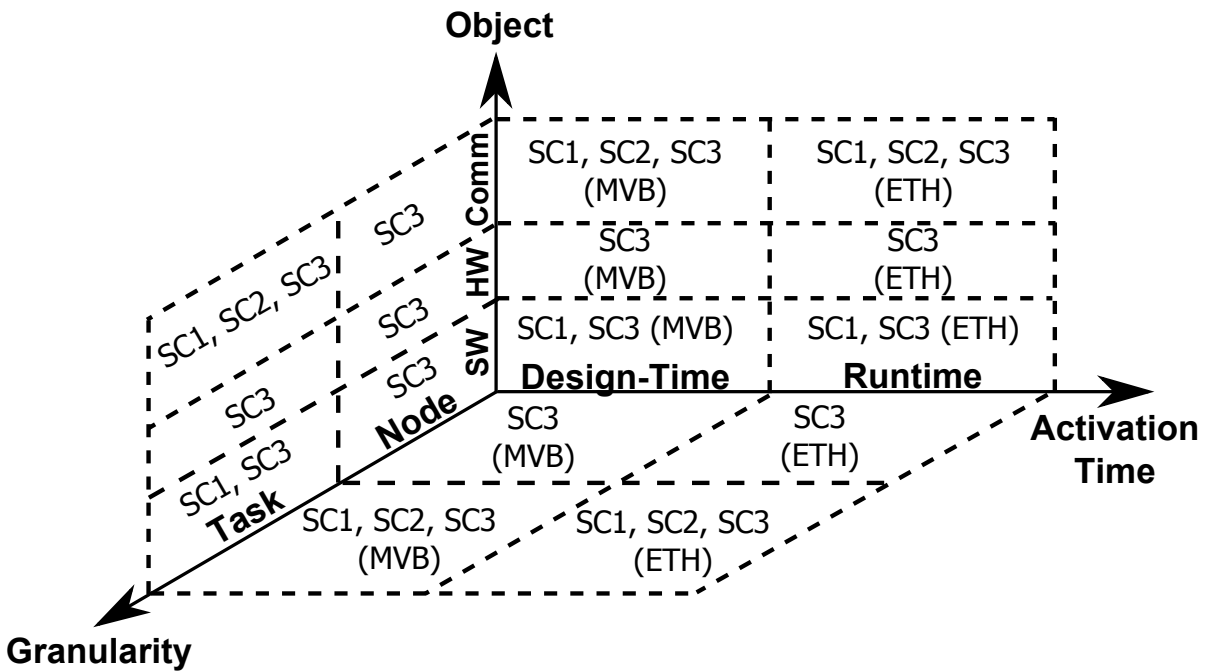


Figure 6.9: Reconfiguration Space of the Tested Scenarios

6.3.1 Scenario I: Sensor-Level Reconfiguration

Without losing the applicability of the scenario, *SC1* focuses on the example presented in [Aizpurua12a] and described in Section 2.1 (see the functional model in Table 3.4).

A train car vehicle may have different compartments (cf. Figure 6.10 $Zone_A$, $Zone_B$) and independent Air Conditioning Control main function implementations at each compartment. Assume that 2 PUs are connected to perform Air Conditioning Control in each vehicle's compartment: one PU (PU_1 or PU_3) measures the temperature (SF1: temperature measurement) using a sensor (S_1 or S_2) and gets the reference temperature (SF2: user reference temperature) using a reference knob (R_1 or R_2), and the second PU (PU_2 or PU_4) acts as a controller (SF3: air conditioning control algorithm) and gives the output to the connected heater (H_1 or H_2).

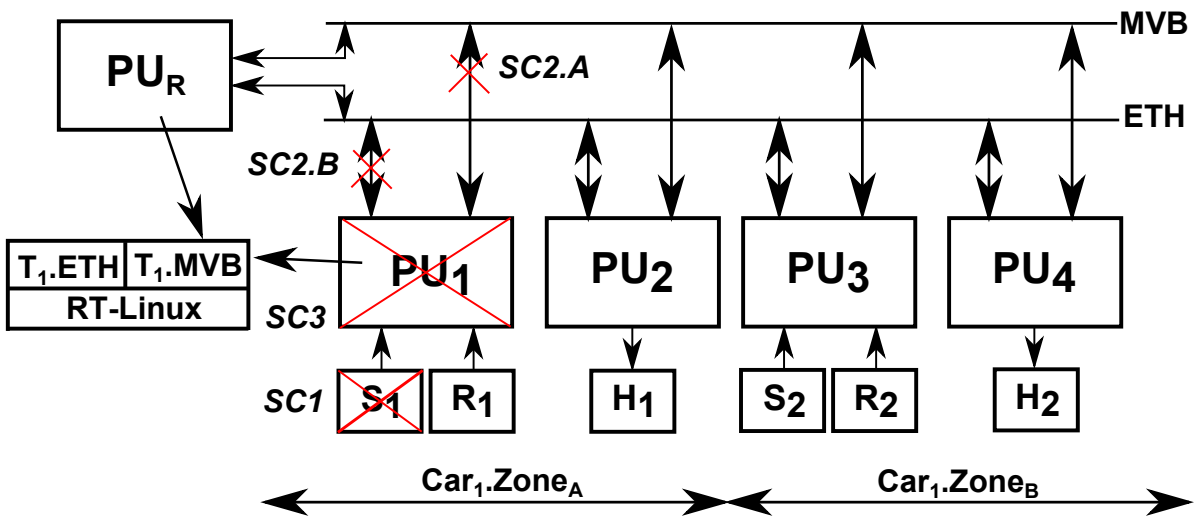


Figure 6.10: Reconfiguration Scenarios

Let us focus on the reconfiguration of temperature measurement subfunction at $Car_1.Zone_A$. The nominal communication *Route* of the temperature measurement subfunction in each compartment is as follows:

$$Route_1: S_1 \rightarrow PU_1 \rightarrow ETH \rightarrow PU_2 \rightarrow H_1;$$

$$Route_2: S_2 \rightarrow PU_3 \rightarrow ETH \rightarrow PU_4 \rightarrow H_2.$$

Given that one sensor of any compartment fails, we reuse the already existing one in the same car, but in a different compartment. To reconfigure the temperature measurement implementation in $Car_1.Zone_A$ its value-based fault detection is located in the destination processing unit PU_2 . When sensor S_1 fails, incorrect or missing values are detected at PU_2 by the fault detection, and the reconfiguration implementation orders the faulty component to stop sending data. It also checks the IP address and the User Datagram Protocol (UDP) port of the next standby implementation of tem-

perature measurement subfunction in its reconfiguration table, and it establishes the communication with S_2 . This process changes the communication route from $Route_1$ to $Route_{12}$:

$$Route_{12}: S_2 \rightarrow PU_3 \rightarrow ETH \rightarrow PU_2 \rightarrow H_1.$$

The design of the devices identified as heterogeneous redundancies enables them to redirect their information to different information sinks dynamically when a reconfiguration signal is received. During the reconfiguration, source and sink PUs synchronize and S_2 continues sending data towards PU_2 until S_1 is repaired and reconfigured. Implemented reconfiguration mechanisms are applicable to input subfunction implementations operating with heterogeneous redundancies (e.g., Fire Protection Control example cf. Table 3.8). MVB reconfigurations apply the same process, with the difference that $Route_{12}$ is activated from the outset.

6.3.2 Scenario II: PU-Level Reconfiguration

Since a train incorporates different communication protocols, there is room to benefit from heterogeneous redundant communications. Despite bidirectional communications have been implemented between PU_1 and PU_2 , for simplicity the following unidirectional *Routes* are considered:

$$Route_1: T_1.MVB \rightarrow PU_1 \rightarrow MVB \rightarrow PU_2;$$

$$Route_2: T_1.ETH \rightarrow PU_1 \rightarrow ETH \rightarrow PU_2.$$

Where $T_1.MVB$ and $T_1.ETH$ identify MVB and Ethernet tasks respectively (cf. Figure 6.10). When a communication link is down, the general communication-level reconfiguration process is as follows:

1. The application located in the destination PU detects the communication failure (time-based fault detection).
2. Subsequently, it reconfigures itself creating a server to continue receiving data using the operating communication protocol.
3. It informs the source PU about the communication failure.

4. Finally, the source PU is also reconfigured switching from the faulty to the operating communication

Hence, when MVB is disconnected (SC2.A, cf. Figure 6.10), UDP communication threads are created dynamically to continue sending MVB data via Ethernet changing communication routes from $Route_1$ to $Route_{12}$ where,

$$Route_{12}: T_1.MVB \rightarrow PU_1 \rightarrow ETH \rightarrow PU_2.$$

And vice versa, when Ethernet is disconnected (SC2.B, cf. Figure 6.10) the communication route is changed from $Route_2$ to $Route_{22}$ where,

$$Route_{22}: T_1.ETH \rightarrow PU_1 \rightarrow MVB \rightarrow PU_2.$$

6.3.3 Scenario III: Communication-Level Reconfiguration

Point to point unidirectional communication from PU_1 to PU_2 is considered with the next communication routes:

$$\begin{aligned} Route_1: T_1.MVB &\rightarrow PU_1 \rightarrow MVB \rightarrow PU_2; \\ Route_2: T_1.ETH &\rightarrow PU_1 \rightarrow ETH \rightarrow PU_2. \end{aligned}$$

The tasks that PU_1 is performing are rearranged in another compatible PU to deal with the failure of PU_1 . A higher level reconfiguration implementation (PU_R) has been added to redirect all the data that the failed PU was sending from its communication interfaces. PU_R monitors the performance of both PUs (PU_1 , PU_2) and when it detects that any of them is down (time-based fault detection); it is reconfigured sending the data that it was sending before through MVB and Ethernet. Consequently, $Route_1$ is replaced by $Route_{12}$ and $Route_2$ switches to $Route_{22}$ where,

$$\begin{aligned} Route_{12}: T_1.ETH &\rightarrow PU_R \rightarrow ETH \rightarrow PU_2; \\ Route_{22}: T_1.MVB &\rightarrow PU_R \rightarrow MVB \rightarrow PU_2. \end{aligned}$$

6.4 Conclusions

In this chapter we have presented a real HW/SW architecture, that based on industrial railway communication devices implements the ideas treated throughout this dissertation.

The architecture have been tested on different scenarios to validate the system's fault tolerance capabilities under different failure situations. Namely, we have analysed the architecture with respect to sensor, communication and PUs failures by reusing already existing elements.

The main limitation of the experiments carried out in this chapter is that the scenarios have been tested isolated from the other functions comprising a real train. Hence, we do not have to deal with possible memory and bandwidth issues. A more accurate approach would require taking into account these requirements as well as performing calculations so that the system meets all its requirements.

Conclusions and Future Work

In this chapter we present the main results and limitations of this dissertation. The chapter is organised as follows:

- Section 7.1 summarizes the work performed during the completion of the research work.
- Section 7.2 points out the outcomes obtained from this dissertation.
- Section 7.3 identifies the limitations of the research work and future research areas which deserve attention to further improve this work.

7.1 Conclusions

This dissertation comprehends multiple engineering fields including systems engineering, software engineering and reliability engineering. The main contributions of this dissertation are confined in the design of reconfigurable systems considering optimization of design decisions with respect to dependability and cost.

During this research period a comprehensive review and classification of dependability analysis, verification and design approaches has been performed [[Aizpurua12b](#)] [[Aizpurua13b](#)]. These papers review the state of the art approaches in the field of model-based dependable design including dependability analysis and verification approaches. They point out advantages and disadvantages of the well known event-based and state-based approaches and accordingly, the approaches from the scientific literature are classified based on the addressed limitations.

We have designed the D3H2 (aDaptive Dependable Design for systems with Homoge-

neous and Heterogeneous redundancies) methodology with the goal of optimising design decisions in massively networked scenarios. The methodology enables the evaluation of the influence of the different design decisions on dependability and cost, including the reuse of existing resources. It also aids the designer to choose between redundancy and reconfiguration strategies. Chapter 3 overviews the D3H2 methodology and characterizes the key modelling and analysis activities to design a HW/SW architecture taking into account its cost [Aizpurua13a]. Namely, the Functional Modelling Approach (cf. Subsection 3.3.1) and the Compatibility Analysis (cf. Subsection 3.3.2) enable the systematic identification of redundancies and single points of failure [Aizpurua12a]. The Extended Functional Modelling Approach (cf. Subsection 3.3.4) enables the systematic extension of the initial HW/SW architecture and allows the designer to create the *extended HW/SW architecture* which accounts for design decisions with respect to the distribution and implementation of fault detection, reconfiguration and communication functions.

Chapter 4 presents the Dependability Evaluation Modelling approach for non-repairable systems. This approach is used to perform a complete and systematic assessment of the *extended HW/SW architectures* and evaluate the influence of alternative architectural design decisions on dependability. The component-based nature of the dependability evaluation algorithm enables to perform the probabilistic analysis of the Dependability Evaluation Model using the combination of Dynamic Fault Tree and Component Fault Tree approaches, that is, *Component Dynamic Fault Trees* [Aizpurua14].

The outlined approach makes it possible to evaluate the effect of ideal/non-ideal health management and communication implementations on the system failure probability using importance measurements. This approach can be exploited to analyse the contribution of these implementations to the system's failure probability. Furthermore, in order to deal with the lack of exact failure data information of some resources (e.g., software resources), uncertainty analyses have been implemented. Therefore, it is possible to specify interval failure rates (instead of single value data) of system resources and calculate the failure probability distribution of the top event's failure probability, i.e., second order failure probabilities.

When analysing non-repairable systems, the evaluation of which redundancy strategy is cheaper does not have only one answer. Depending on the type of the heterogeneous redundancy strategy their costs are different. Generally speaking, het-

erogeneous redundancies arising from natural compatibilities require less additional resources than heterogeneous redundancies arising from forced compatibilities and they are usually more cost-effective.

Chapter 5 defines the Dependability Evaluation Modelling approach for repairable systems. It gives methods to assess exhaustively and systematically the influence of alternative architectural design decisions on dependability. Compared with non-repairable systems, the Dependability Evaluation Modelling approach for repairable systems requires more powerful formalisms for considering random failure and repair sequences. Based on the Stochastic Activity Networks formalism, we have implemented the Dependability Evaluation Modelling approach that takes into account complex repair strategies.

Depending on the design-specific decisions for each main function, the influence on system's dependability and cost varies. As confirmed in Chapter 4 and Chapter 5, optimisation of design decisions with respect to the type and number of redundancy and reconfiguration strategies to maximize dependability and minimize the cost are feasible.

The influence of redundancies on system dependability and cost depend on the analysed main function and its configuration. There are different variables that influence dependability and cost. Concerning the dependability we identify the next factors:

- Number of redundancy implementations: the greater the number of redundancies, the lower the failure probability of the subfunction.
- Type of redundancy implementations: generally speaking the failure probability of the heterogeneous redundancies is greater than homogeneous redundancies due to the added extra resources in order to make implementations compatible (e.g., SW resources, communication).
- Number of reconfiguration's redundancies: its increase contributes directly to the improvement of the reconfiguration subfunction's performance. However, its contribution to the reduction of the failure probability of the main function is almost negligible because is attenuated by sequence-dependent intermediate failure events.
- Type of reconfiguration strategy: distributed reconfiguration redundancies have shown a lower failure probability of the reconfiguration subfunction than the centralised reconfiguration redundancies which concentrate in a single processing unit

all the function redundancies.

- Communication: if the whole system is connected using the same communication network it becomes a critical factor. In these cases it cannot be assumed as ideal and its contribution to the top-event failure should be evaluated.

Generally heterogeneous redundancies obtained from natural compatibilities require less resources to make implementations compatible. In these configurations the failure probability difference between homogeneous and heterogeneous redundancies is lower compared with heterogeneous redundancies obtained from forced compatibilities.

In a main function, the failure contribution of a subfunction without redundancies is more important than the contribution of a subfunction with redundancies. That is, the number of redundancies and the number and type of reconfiguration redundancies of a subfunction become less effective when the same main function has another subfunction which has a single implementation (i.e., a single point of failure).

The following are the main factors that influence system cost:

- Downtime cost: the influence of the downtime cost is higher with less reliable architectures and it is more penalising than the cost incurred by HW, SW or communication resources.
- Type of redundancies: the architecture cost (HW, SW and communications resource cost) of heterogeneous redundancies are cheaper than homogeneous redundancies, however, when downtime costs are included, the cost depends on the system's unavailability - which is commonly better for homogeneous redundancies.
- Type of heterogeneous redundancies: heterogeneous redundancies arising from natural compatibility does not need a specific software, whereas heterogeneous redundancies arising from forced compatibility requires fit-for-purpose software.
- Number of heterogeneous redundancies: if there are similar heterogeneous redundancies arising from forced compatibilities, the cost of each heterogeneous redundancy is lower. This happens because the software development cost of one software resource (which is assumed to be valid for all redundancies with slight modifications) is divided among the similar heterogeneous redundancies. Therefore, the cost per each heterogeneous redundancy is not as

high as for a single (independent) heterogeneous redundancy (related to SW development cost attribution).

Note that our analysis have been performed for the values shown in Appendix E. In order to contrast the validity of the obtained results a cost sensitivity analysis should be performed.

The reuse of system resources (i.e., heterogeneous redundancy) reduces system cost compared with the addition of an additional hardware components. However, this is only true when the unavailability incurred by the heterogeneous redundancy is not greater (or is slightly greater) than the homogeneous redundancy.

Depending on the system configuration, the influence of health management and communication implementations on system dependability may be negligible or not. In the following some deliberations about health management implementations:

- Depending on the number of input, control or output subfunction's redundancies, the influence of health management implementations gains significance. The less subfunctions, the higher its weight (e.g., a single input subfunction with redundancies).
- The higher the number of redundancies of health management implementations, the lower the failure probability of the reconfiguration and fault detection implementations and higher the system cost. However, its effect on the main function is attenuated due to the sequence-dependent constraint (health management implementations must fail prior to the subfunction's implementation).
- If the implementations of the same subfunction are concentrated in a single PU, the system becomes more sensitive to communication failures.

The feasibility of the use of heterogeneous redundancies for safety-critical functions is an issue worth mentioning. In some cases, the cost incurred in obtaining evidences of the reliability of the heterogeneous redundancy can increase the cost more than using an homogeneous redundancy.

7.2 Contributions

The followings are the main contributions of this dissertation:

- A comprehensive review of the model-based dependability analysis, verification and design approaches has been developed.
- We have developed a methodology that enables the systematic characterization and evaluation of HW/SW architectures which includes:
 - Systematic identification of heterogeneous redundancies.
 - Systematic evaluation of the influence of design decisions on system dependability for non-repairable systems. In this context, an analysis paradigm that allows the transformation of the design model to the dynamic dependability analysis model has been used (Component Dynamic Fault Trees).
 - Systematic evaluation of the influence of design decisions on system dependability for repairable systems. This approach enables the analysis of the failure probability of the system taking into account prioritized repair strategies and including components with complex logic and repeated events.
- So far, the research community has considered health management implementations as ideal when using heterogeneous redundancies. Our methodology includes health management mechanisms (and their failure model) as well as homogeneous and heterogeneous redundancies when designing adaptive dependable systems.
- Hitherto, heterogeneous redundancies have not been integrated in a design methodology that starts from their identification, moves through the construction of the HW/SW architecture to use them in massively networked scenarios, and quantifies their effect on system's dependability and cost.
- Validation of the concepts treated throughout the dissertation using industrial railway communication devices.

7.3 Future Work

The goals of this work have been focused on the stated research objectives. However, there exist some interesting areas that have not been completed during this time and they deserve to be mentioned in order to progress in the use of heterogeneous redundancies in real systems. Subsequently, we list some points that determine how this thesis can be further developed.

The presented modelling approaches (Functional Modelling Approach and Extended Functional Modelling Approach) enable a straightforward characterization of the system and its subsequent exploitation for redundancy identification and further analyses. However, this process requires studying all the system functions, resources, and their physical locations early at the design time. In order to alleviate the burden of annotations it may be possible to come up with an approach that enables the auto-annotation (suggestion) of implementations based on the components name.

When using heterogeneous redundancies, the designer needs to be aware of the quality degradation and evaluate whether it is acceptable or not. To further refine the compatibility analysis, heterogeneous redundancies should be validated exhaustively. To this end, different architecture-specific requirements subject to real system operation need to be taken into account, such as acceptable error margins, timeliness, memory and processing capacity constraints of the processing units.

Another issue worth addressing is the construction of the reconfiguration table at run-time. Run-time updates to the reconfiguration table would facilitate the system maintenance and it would reflect the real system status.

The approach would benefit from the automatic extraction of the dependability evaluation models so that the designer is not exposed to error-prone tasks. Besides, the automation would help to implement optimization algorithms in order to search for the best architecture according to predefined dependability and cost requirements.

The time needed to carry out the simulations of the dependability analysis models is considerable: this issue originates from the level of complexity of the dependability evaluation model and the required accuracy of the results. In this direction, techniques such as dynamic simulation stopping criterion can be defined: deciding whether to

continue simulating the model based on simulation parameters (e.g., acceptable standard deviation of the probability calculations).

In the analysed case studies the evaluations have been carried out at the function level. However, it could be interesting to consider the train car as a whole. In this way, an overall evaluation of the train's performance could be evaluated while considering all the performed functions simultaneously.

As for the cost assessment, undertaking a cost sensitivity analysis would consolidate the conclusions that we have obtained in this dissertation.

When validating the concepts treated throughout the dissertation using industrial communication elements, the main limitation has been that the scenarios have been tested isolated from the other functions comprising a real train. Hence, we do not have to deal with possible memory and bandwidth issues. A more accurate approach would require taking into account these requirements as well as performing calculations so that the system meets all its requirements.

In Chapter 5, we assumed that (1) the repair process of resources starts as soon as a resource fails and (2) the repaired resource is as good as a new one. In this direction, there may be some points worth analysing:

- It may be possible to implement preventive maintenance strategies so that a component is repaired/replaced before its failure.
- The degradation of the resource after reparation can be taken into account, e.g., increasing the failure rate after each reparation.
- It would be interesting to evaluate the influence of repair rates on system's failure probability to optimize repair and maintenance parameters.

Appendices

Overview of the Basic Dependability Analysis Approaches

A.1 Event-Based (Combinatorial, Static) Approaches

Event-based approaches characterize the system failure behaviour through the combination of its constituent components failure events. This characterization reflects system's structural properties (e.g., redundancies), but it is unable to capture complex events and dependencies. The main advantage of these approaches is their simplicity which has resulted in their widespread use in different industry fields such as railway, avionics or nuclear industries. In contrast, among their disadvantages it should be highlighted that they are unable to grasp system's *dynamics* such as load-sharing, standby redundancies or dependencies. Their underlying (limiting) **assumptions** are the followings:

1. Events are characterized as stochastically independent events.
2. Events are characterized as binary events: working or failed.
3. Non-repairable events: when events fail for the first time, they are assumed to be failed forever.
4. Characterization of a single failure/functioning event at a time.
5. Relations between events expressed by (static) boolean operators.

In the scientific literature there has been proposed many approaches to overcome the limitations of these approaches (see [Aizpurua13b] for an overview of limitations and solutions). Some of them are addressed in the Subsection 2.3.1.

Event-based approaches characterize the failure or functioning logic of the system through the *structure function* of the system [Rausand03]:

A.1.1 Structure Function

Consider a system composed of n components, where the state of the component i (x_i), $i=1, 2, \dots, n$ can be functioning or failed:

$$x_i = \begin{cases} 1, & \text{if component } i \text{ is functioning} \\ 0, & \text{if component } i \text{ is in a failed state} \end{cases}$$

$\mathbf{x} = (x_1, x_2, \dots, x_n)$ is called the *state vector*. The state of a system can be described by a binary function:

$$\Phi(\mathbf{x}) = \Phi(x_1, x_2, \dots, x_n) \tag{A.1}$$

where

$$\Phi(\mathbf{x}) = \begin{cases} 1, & \text{if the system is functioning} \\ 0, & \text{if the system is in a failed state} \end{cases}$$

and $\Phi(\mathbf{x})$ is called the *structure function* of the system. *Series*, *parallel*, and *K out of N* structure are the classical arrangements of systems with the following structure functions:

Series Structure: a system that is functioning if all of its n components are functioning:

$$\Phi(\mathbf{x}) = x_1 \cdot x_2 \cdot \dots \cdot x_n = \prod_{i=1}^n x_i \tag{A.2}$$

Parallel Structure: a system that is functioning if at least one of its n components is functioning:

$$\Phi(\mathbf{x}) = 1 - (1 - x_1) \cdot (1 - x_2) \cdot \dots \cdot (1 - x_n) = 1 - \prod_{i=1}^n (1 - x_i) \quad (\text{A.3})$$

K out of N Structure: a system that is functioning if at least k of the n components are functioning:

$$\Phi(\mathbf{x}) = \begin{cases} 1, & \text{if } \sum_{i=1}^n x_i \geq k \\ 0, & \text{if } \sum_{i=1}^n x_i < k \end{cases} \quad (\text{A.4})$$

Making use of the structure function, we will define two well known event-based dependability analysis approaches: Fault Tree Analysis and Reliability Block Diagrams.

A.1.2 Fault Tree Analysis

The concept of Fault Tree Analysis (FTA) was developed by Bell Telephone Laboratories as a technique with which to perform a safety evaluation of the Minuteman Launch Control in 1961. Later Boeing¹⁵ company modified it for computer utilization and now it is widely used in many fields such as aviation, railway or nuclear [Office02].

FTA is a top-down deductive analysis technique aimed at finding all the ways in which a failure can occur. Starting from an undesirable system-level failure, i.e., top-event, its immediate causes to occur are identified until reaching the lowest-level component, i.e., basic-event. The top-event is broken down into intermediate and basic-events linked with logic gates organised in a tree-like structure. The resulting FT, is a model in the form of combinations of events which are necessary to the top-event to occur. The combination of events are specified using boolean logic gates denoting the relationship between the different events (see Figure C.1 for a FT model example). Formally,

Definition A.1. Fault Tree (FT): A fault tree model, ft , is defined by a 4-tuple: $ft = \langle TE, BE, BG, R \rangle$

where:

- TE is the top-event of the FT (failure of the modelled system)

¹⁵www.boeing.com

- BE is the set of basic events
- $BG = \{AND, OR, KooN\}$ is the set of boolean gates
- $R \subseteq (BE \times BG) \cup (BG \times BG) \cup (BG \times TE)$ is the set of relations.

Often the output of a gate which is then connected to another gate is named *Intermediate Event (IE)*. Boolean Gates (BG) are defined as follows:

- **AND:** $Y = AND(E_1, E_2, \dots, E_N)$; Y is true iff all events $\{E_1, E_2, \dots, E_N\}$ are true; otherwise is false (cf. Figure A.1 (b)).
- **OR:** $Y = OR(E_1, E_2, \dots, E_N)$; Y is true iff any event $\{E_1, E_2, \dots, E_N\}$ is true; otherwise is false (cf. Figure A.1 (c)).
- **KooN:** $Y = KooN(E_1, E_2, \dots, E_N)$; Y is true iff at least k ($1 < k < n$) among the set of N input events $\{E_1, E_2, \dots, E_N\}$ is true; otherwise is false (cf. Figure A.1 (d)).

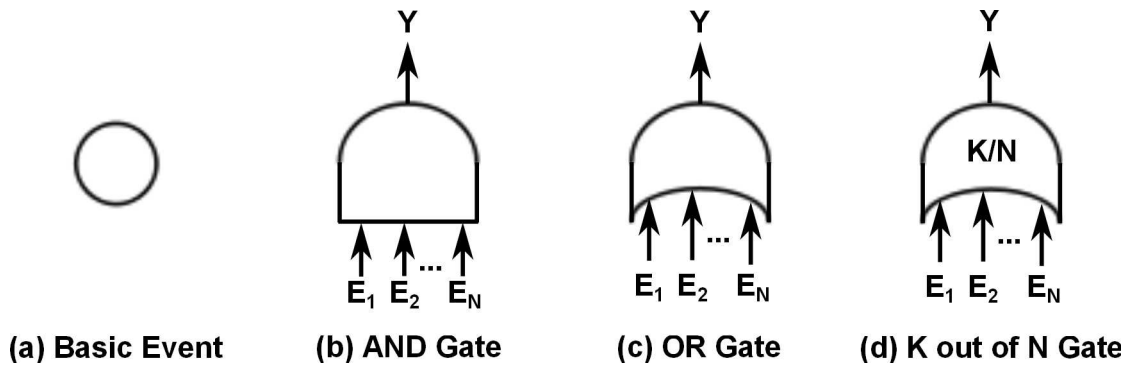


Figure A.1: Fault Tree Symbols

Qualitative Analysis: the principal qualitative results are the (minimal) cut-sets, which reflect the (smallest) combination of basic events whose simultaneous occurrence results in the top-event occurrence. The number of possible cut-sets grows exponentially with the size of the fault tree.

Quantitative Analysis: A FT model can be quantified by ascribing probabilities to the basic events and combining them to evaluate the probability of the top-event:

- **Structure Function:** replace system variables with the corresponding failure probability.

- Computation based on Minimal Cut-Sets (MCS): determine all the minimal cut-sets $MCS_1, MCS_2, \dots, MCS_k$, and rewrite the structure function as follows:

$$\Phi(\mathbf{x}) = \prod_{j=1}^k \prod_{i \in MCS_j} x_i \quad (\text{A.5})$$

From the resulting structure function, once system variables are replaced with the corresponding probabilities, the inclusion-exclusion formula should be applied to determine the system unreliability and avoid taking into account probabilistic dependency between events.

Importance Measurements: Importance measurements can be carried out to quantify the contribution of the BE (or IE) occurrences to the TE failure. There exist different importance measurement methods based on the influence of the (1) BE's (or IE) reliability and (2) structural location of the BE (or IE) in the system. Different importance measurements have been defined based on these properties, refer to Section 4.3 for further details and references.

Binary Decision Diagram (BDD) based Analysis [Bryant86]: BDD encodes the boolean formula underlying a FT model. It allows the reduction of the fault tree by providing advantages from computational point of view. Working directly in the logical expression level it allows to obtain minimal cut-sets and system level unreliabilities.

The BDD approach is based on the Shannon decomposition formula [Shannon38] and its equivalent *if-then-else* (ite) structure:

$$F = x_1 \wedge F_1 \vee \bar{x}_1 \wedge F_0 = ite(x_1, F_1, F_0) \quad (\text{A.6})$$

That is, if x_1 is true then F_1 else F_0 . For instance, if we consider the next boolean formula, which expresses the failure logic of a simple system: $TE = x \wedge y \vee z$; the TE failure logic can be expressed as follows: $TE = ite(x, ite(y, 1, ite(z, 1, 0)), ite(z, 1, 0))$. Accordingly, the corresponding BDD which encodes the boolean formula into *ite* notation is shown in Figure A.2.

There exist many algorithms and tools for the synthesis, optimization, verification and testing of BDDs [Doyle95].

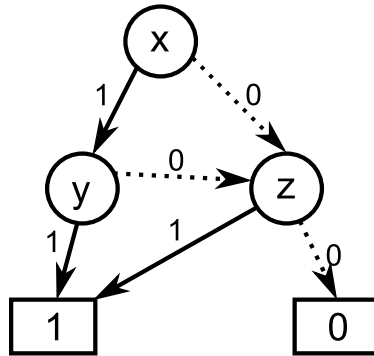


Figure A.2: The BDD of the formula $y = x \wedge y \vee z$

A.1.3 Reliability Block Diagrams

A Reliability Block Diagram (RBD) is a success-oriented network describing the non-repairable function of the system [Rausand03]. It shows the logical connections of the components needed to fulfil a specified system function answering the following question: which elements of the item under consideration are necessary for the fulfilment of the required function and which can fail without affecting it? [Alessandro06]

Each component is illustrated by a block (*reliability block*) and each of them has its specific failure characteristics. Blocks are combined as series structure, parallel structure or K-out-of-N structure to fulfil the specified system function according to the redundancy scheme (see Figure A.3 for some example configurations). Formally,

Definition A.2. Reliability Block Diagram (RBD): A reliability block diagram model, rbd , is defined by a 4-tuple: $rbd = \langle B, C, N, J \rangle$ where:

- B is the set of blocks
- C is the set of connections between the blocks
- N is the set of nodes
- $J \subseteq (N \times C \times B) \cup (B \times C \times N) \cup (B \times C \times B)$ is the connection relation with respect to the input node, output node; and the connection relation between blocks respectively.

Qualitative Analysis: the principal qualitative results are the (minimal) path-sets,

which reflect the (smallest) combination of blocks whose simultaneous occurrence result in the correct operation of the system.

Quantitative Analysis: A RBD model can be quantified by ascribing probabilities to the blocks and combining them to evaluate the probability of the system functioning or failing:

- **Structure Function:** in the structure function replace the system variables with the corresponding working probability.
- **Computation based on Minimal Path-Sets (MPS):** determine all the minimal path-sets $MPS_1, MPS_2, \dots, MPS_k$, and rewrite the structure function as follows:

$$\Phi(\mathbf{x}) = \prod_{j=1}^k \prod_{i \in MPS_j} x_i \quad (\text{A.7})$$

From the resulting structure function, once system variables are replaced with the corresponding probabilities, the inclusion-exclusion formula should be applied to determine the system reliability and avoid taking into account overlapping events.

A.2 State-Based (Dynamic) Approaches

State-based approaches make use of state-space models to quantify RAMS properties of the system under study. They characterize the occurrence of a failure as a transition from functional state to a failed state. That transition can be provoked either by another event which triggers the state change or due to the elapsed time in a state. State-based analysis techniques mainly differ in their abstraction levels and considered probabilistic distributions.

The advantages of the state-based approaches over event-based approaches are that they account for the reliability of system's *dynamics*. However, their disadvantage are the complexity to analyse accurately system's dependability properties and the state-explosion problem.

The following paragraphs introduce some relevant definitions to characterize state-based approaches:

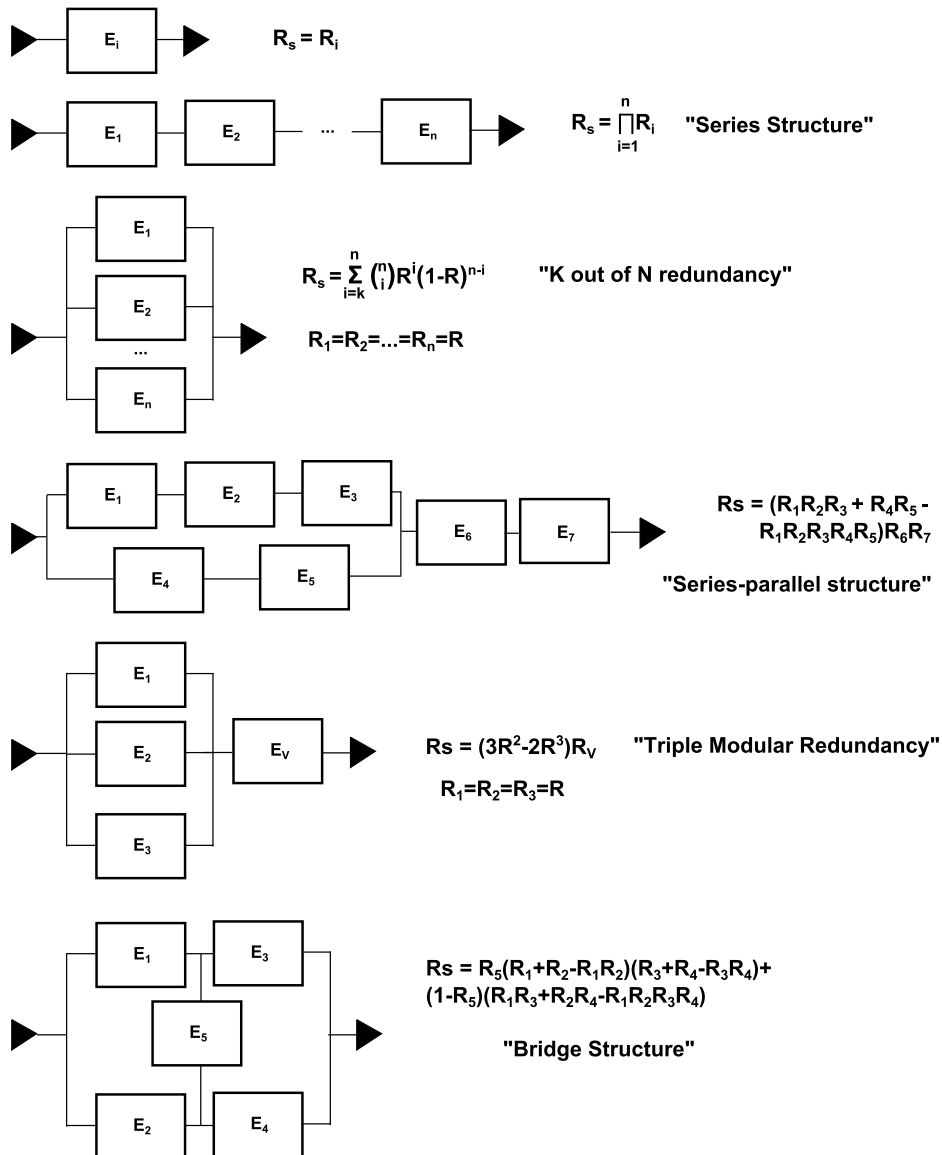


Figure A.3: Reliability Block Diagram Structures and Associated Reliability Functions [Alessandro06]

Let S be the possible outcomes of a random experiment. The set S is called sample space of the experiment. A random variable is the mapping from $s \in S$ a outcome to a real number.

Definition A.3. Stochastic Process: family of random variables $\{X(t) \mid t \in T\}$ defined on a given probability space.

The values of the random variable $X(t)$ denote system states and T is called the pa-

parameter set. If T is not countable the process is said to have a *continuous parameter*; otherwise it is called *discrete parameter process*.

Definition A.4. State Space: the state space Ω of the process $\{X(t)\}$ is determined by the set of all possible values that random variables can take. Depending if T is continuous or discrete, the state space is called *continuous state space* or *discrete state space* respectively.

Two events ($A, B \in F$) are said to be independent if: $P(A \cap B) = P(A)P(B)$. Thus, an independent stochastic process is defined as follows:

Definition A.5. Independent Stochastic Process: assuming that $F_n(x_1, x_2, \dots, x_n)$ denotes the finite dimensional joint distribution of a stochastic process $\{X(t) \mid t \in T\}$, the stochastic process is independent if:

$$F_n(x_1, \dots, x_n) = P\{X(t_1) \leq x_1, \dots, X(t_n) \leq x_n\} = \prod_{i=1}^n P\{X(t_i) \leq x_i\} \quad (\text{A.8})$$

Among the state-based approaches, we will focus on two well known basic approaches from which different approaches have come up: Markov Chains and Petri Nets.

A.2.1 Markov Chains

Markov Chain based analysis techniques describe states of a system at successive times [Haverkort01; Trivedi02]. The Markov property states that the system depends only on the current state and not on the history of the states:

Definition A.6. Markov Property: if for any $t_0 < t_1 < t_2 < \dots < t_n < t$, the conditional distribution of $X(t)$ for given values of $X(t_0), X(t_1), X(t_2), \dots, X(t_n), X(t)$ depends only on $X(t_n)$:

$$\begin{aligned} P\{X(t) \leq x \mid X(t_n) = x_n, \dots, X(t_1) = x_1, X(t_0) = x_0\} \\ = P\{X(t) \leq x \mid X(t_n)\} \end{aligned} \quad (\text{A.9})$$

The Markovian property is also known as *memoryless property* and a stochastic process

which possesses the Markov property is called a *Markov process*. A Markov process with a discrete state space is referred to as a *Markov Chain*.

In most Markov processes it is normal to assume that they are time invariant or time homogeneous¹⁶ (also known as stationary Markov Chains) satisfying:

$$P\{X(t) \leq x \mid X(t_n)\} = P\{X(t - t_n) \leq x \mid X(0) = x_n\} \quad (\text{A.10})$$

The probability that the process stays in state i at time $t > t_n$ given it was in state i at time t_n only depends on state i , but does not depend on how much time it has spent in state i (no state age memory). This property implies that the lifetimes between subsequent events should also have the memoryless property, i.e., sojourn time is exponentially distributed for continuous processes and geometrically distributed for discrete processes.

Depending on continuous or discrete transition times between states, Markov chains are classified as *Continuous Time Markov Chain (CTMC)* (cf. Figure A.4) or *Discrete Time Markov Chain (DTMC)* (cf. Figure A.5) respectively:

Definition A.7. Continuous Time Markov Chain (CTMC): A Continuous Time Markov Chain model, *ctmc*, is a 3-tuple: $ctmc = \langle S, s_0, R \rangle$ where:

- $S = \{s_0, s_1, s_2, \dots, s_p\}$ is a finite set of states;
- $s_0 \in S$ is the initial state;
- $R : S \times S \rightarrow \mathbb{R}_{\geq 0}$ is the transition rate matrix.

The transition time (sojourn time or delay) is characterized according to the exponential distribution. The transitions in DTMCs are labelled with probabilities instead of rates:

Definition A.8. Discrete Time Markov Chain (DTMC): A Discrete Time Markov Chain model, *dtmc*, is a 3-tuple: $dtmc = \langle S, s_0, R \rangle$ where:

- $S = \{s_0, s_1, s_2, \dots, s_p\}$ is a finite set of states;

¹⁶In the following we consider only time-homogeneous Markov chains

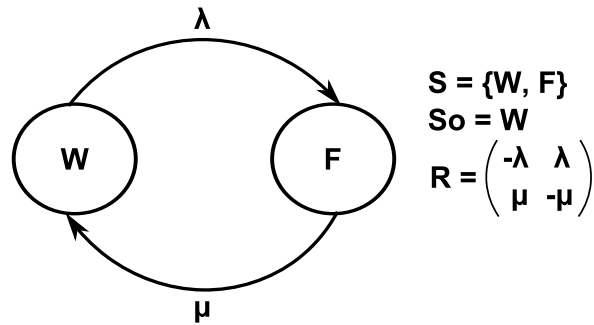


Figure A.4: Continuous Time Markov Chain Example

- $s_0 \in S$ is the initial state;
- $R : S \times S \rightarrow [0, 1]$ is the transition probability matrix.

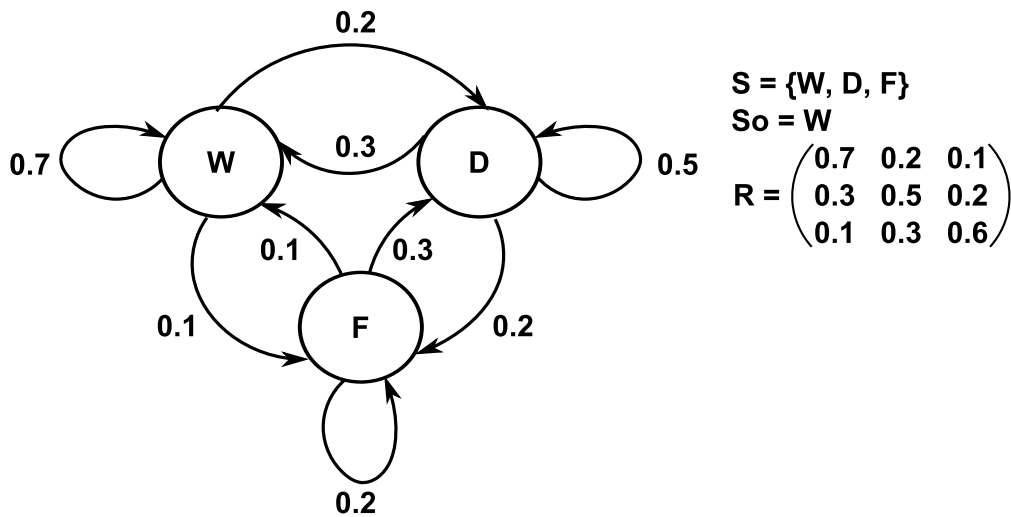


Figure A.5: Discrete Time Markov Chain Example

With homogeneous continuous Markov Chains, the sojourn time is exponentially distributed, but in some cases this is not enough to describe system's properties adequately. Besides, Markov Chains are modelled as flat networks, thus, when dealing with complex systems, the readability and correct construction of the Markov models is complicated and in many cases it suffers from the state explosion problem. In this way, stochastic extensions were introduced to alleviate the complexity of the pure Markov Chain models when characterizing complex systems.

Definition A.9. Renewal process: let $S_0 < S_1 < S_2 < \dots$ be the time instants of successive events to occur where,

$$S_0 = 0 \text{ and } S_n = \sum_{i=1}^n X_i \quad n = 1, 2, \dots \quad (\text{A.11})$$

The sequence of non-negative independent and identically distributed random variables $S = \{S_n - S_{n-1}; n = 1, 2, \dots\}$ is a renewal process; i.e., the sequence of interoccurrence times between successive events are independent and identically distributed.

The state at S_n (the epoch that the n -th event occurs) is given by $X_n \in S$. The chain X_n now forms a process on its own (DTMC). The points $S_n; n = 0, 1, 2, \dots$ are called Markov regeneration epochs or Markov renewal moments. Together with the X_n they define a Markov renewal sequence:

Definition A.10. Markov Renewal Sequence: a sequence of bivariate random variables $\{(Y_n, S_n), n \geq 0\}$ is called a Markov Renewal Sequence if:

$$\begin{aligned} &P\{Y_{n+1} = j, S_{n+1} - S_n \leq x \mid Y_n = i, S_n, Y_{n-1}, S_{n-1} \dots, Y_0, S_0\} \\ &= P\{Y_{n+1} = j, S_{n+1} - S_n \leq x \mid Y_n = i\} \quad (\text{Markov property}) \\ &= P\{Y_1 = j, T_1 \leq x \mid Y_0 = i\} \quad (\text{Time Homogeneity}) \end{aligned} \quad (\text{A.12})$$

In a Markov Renewal Sequence, the future evolution of the stochastic process depends on the current state of the process at Markov renewal points, i.e., at time epochs S_n . Markov Renewal Sequences are embedded into Markov Renewal Models. Markov Renewal Models can be classified into two categories [Xie04]: semi-Markov model and Markov regenerative model.

Definition A.11. Semi-Markov process: consider a Markov renewal sequence $\{Y_n, S_n\}$ with state space I the stochastic process $\{Y_n, S_n\}$ is called a semi-Markov process with state space I if $Z(t) = Y(n)$ for $t \in [S_n, S_{n+1})$.

In Semi-Markov processes the amount of time spent in each state before a transition to the next state occurs (i.e., inter-occurrence/sojourn time) is an arbitrary random variable that depends on the next state the process will enter, i.e., the inter-occurrence/sojourn time is not required to be exponentially distributed, instead it follows a general distribution. At transition instants a semi-Markov process behaves like a

Markov process: transitions at Markov renewal points from state to state are made like a Markov process.

Definition A.12. Markov regenerative process [Henk C.03]: if there exists a Markov renewal sequence $\{(Y_n, T_n), n > 0\}$ of random variables such that all the conditional finite dimensional distributions of $\{Z(S_n + t), n \geq 0\}$ given $\{Z(u), 0 \leq u \leq S_n\}, Y_n = i\}$ are the same as those of $\{Z(t), t \geq 0\}$ given $Y_0 = i$.

The Markov regenerative process is a generalization of the semi-Markov process: the Markov regenerative process has state changes between S_i and S_{i+1} , while semi-Markov does not.

A.2.2 Petri Nets

Petri Net approach overcomes the main drawback of the Markov Chain analysis, i.e., the model does not increase in size as the number of components increases. While in a Markov Chain it is necessary to define all the possible combinations of the system, in Petri Nets it suffices with specifying the conditions when a component will be up or down. A Petri Net models the system through the following elements [Peterson81]:

- *Places* which model state variables and contain tokens.
- *Tokens* which model the specific value of state variables.
- *Transitions* which model activities that can cause state changes.
- *Arcs* which model the interconnections between places and transitions.

A *marking* in a Petri Net is an assignment of tokens to the places of a Petri Net (e.g., the marking of the Petri Net depicted Figure A.6 is: $m(p1)=2, m(p2)=0, m(p3)=1, m(p4)=1$ or $m=(2, 0, 1, 1)$). The number and position of tokens may change during the execution of a Petri Net. The tokens are used to define the execution of a Petri net. Formally:

Definition A.13. Petri Net (PN): A Petri Net model, pn , is a 5-tuple:

$$pn = \langle P, T, I, O, M(0) \rangle$$

where:

- $P = \{p_1, p_2, \dots, p_p\}$ is a finite set of places;
- $T = \{t_1, t_2, \dots, t_t\}$ is a finite set of transitions;
- $I: P \times T \rightarrow N$ is an input function that defines the directed arcs from places to transitions, where N is the set of non-negative integer numbers.
- $O: T \times P \rightarrow N$ is an output function that defines directed arcs from transitions to places.
- $M(0) = \{m_1(0), m_2(0), \dots, m_p(0)\}$ is the initial marking, i.e., the number of tokens within the places.

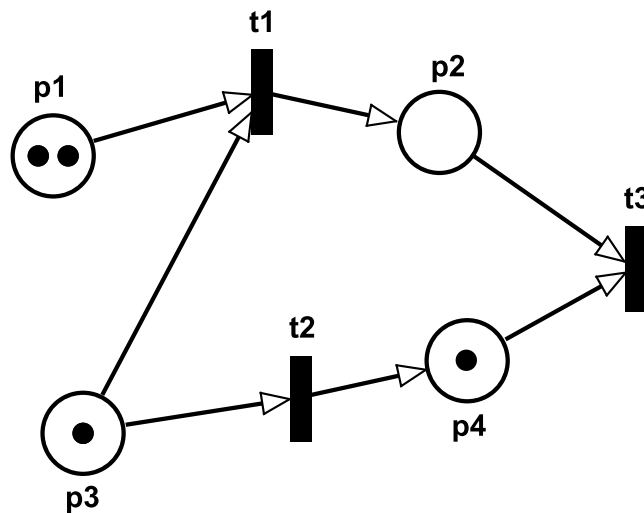


Figure A.6: Petri Net Example

Transitions can be *enabled* when all its input places are marked at least with as many tokens as specified by the input function (e.g., t_1 in Figure A.6). When a transition *fires* it removes the number of tokens from its input places defined by the weight of the input arc and sets to its output place(s) the number of tokens specified by the weight of the output arc. For instance after firing t_1 the resulting marking of the net would be: $m = (1, 1, 0, 1)$.

A Petri Net model simulates the “token game” based on the marking of places. The marking of a Petri Net determines the state of the system. They are used for analysing the probability to reach some desired state.

Originally they were characterized by either deterministic or exponential transition times, which created the mapping between Petri Nets and Markov Chains. As occurred with Markov Chains, the theory of Petri Nets was also extended introducing time dependent transitions:

- Stochastic Petri Nets (SPN): Petri Nets with exponentially timed transitions (or firing delays). Its underlying stochastic process is expressed as CTMC [Bause02].
- GSPN: Petri Nets with exponentially timed and immediate (zero timed) transitions. Immediate transitions have priority over timed transitions. GSPN are also analysed by means of CTMCs [Kartson94].
- DSPN: deterministic (fixed) and exponentially distributed timed transitions [Lindemann98].
- Markov-Regenerative SPN: immediate transitions, exponentially distributed timed transitions and generally distributed (arbitrary) timed transitions [Choi94].
- SAN: generalization of Stochastic Petri Nets, which allows defining general probability distributions and compositional models [Sanders02b] (see Chapter 5).

Petri Nets are high-level representations of the system, which allows (in some cases) the generation of Markov Chain based models. This is why many works analyse systems through Petri Nets-based formalisms, which are characterized by their underlying Markov processes. System's states and events characterized with temporal properties yield to accomplish accurate dependability attributes measurement. Most of the Petri Nets based formalisms characterize the system model with safe-unsafe or working-failed states, where the correctness and accuracy of the analysis depends on the model construction. Petri Nets can also be solved via discrete event simulations [Chiola93b].

Architecture description languages [Medvidovic00] (e.g., UML [OMG14b], AADL [Feiler07]) have been widely adopted to alleviate the dependence on the correctness of the analysis model or quality evaluation model (see Subsection 2.3.1). These approaches include architecture description information as well as dependability behaviour information and automates state-based dependability analysis model generation. However, the expressiveness of the state-based quality evaluation models comes with a considerable computational cost, which is the biggest limitation for state-based approaches, i.e., the state-explosion problem [Valmari98].

Classification of the Hybrid Approaches and Tool Support

The goal of this chapter is to classify the hybrid approaches presented in Chapter 2 and provide information about their tool support. Interested readers please refer to [Aizpurua13b] for more information.

B.1 Classification of the Hybrid Approaches

In order to classify the covered hybrid approaches in Chapter 2, Table B.1 groups them taking into account addressed limitations (see Table 2.5).

Table B.1: Summary of Limitations Overcome by Approaches

Group	Approach	Limitations
1	[Dugan92] [Rao09] [Walter08] [Codetta-Raiteri05] [Montani08] [Manno14c]	L1
2	[Bouissou07] [Manno12b] [Arnold13]	L1, L4
3	[Kaiser03] [Fenelon93] [Domis09b] [Paige08a]	L2, L4
4	[Joshi07] [Adler10a] [Papadopoulos11] [Priesterjahn11a] [Gallina12]	L2, L3, L4
5	[Kaiser07] [Romain07] [Distefano09] [Signoret13] [Niu11]	L1, L2, L4
6	[Walker09] [Montecchi11] [Rugina07] [Riedl12] [Cressent11]	L1, L2, L3, L4

Approaches classified in the group 1 focus on dynamic analysis issues. Differences between them rely in their failure/repair modelling capabilities and their corresponding statistical distributions as noted in Subsection 2.3.1.

Approaches within the groups 2, 3 and 5 allow the compositional evaluation of the system's dynamics (group 3 excepted) addressing the manageability issues arising from the resulting dependability evaluation model.

Approaches gathered within the groups 4 and 6 contain all necessary mechanisms to analyse dynamic systems consistently and in a manageable way. Compositional failure annotation, dynamic behaviour (group 6) and automatic extraction of analysis models are the key features addressed by these approaches. However, when dealing with the manageability and reusability issues (L4) different approaches arise: groups 4 and 6 address L4 by means of the compositional characterization of the design model instead of the compositional characterization of the dependability analysis model. The transformational capability of the design model allows them to cope with design complexity issues. However, the analysis model itself is not a compositional approach, rather it is a flat model whose manageability/maintenance may be hampered when analysing complex systems and dealing with the dependability analysis model directly.

Utilization of failure annotation patterns promote flexibility and reuse and consequently, reduce the error proneness. Nevertheless, as noted in [Lisagor10], characterization of the failure behaviour of components depends on the component context, which conditions compositional and reuse properties. Moreover, automatic generation of the analysis model does not completely alleviate the dependency on the knowledge of the analyst. However, the management and specification of the failure behaviour is clearer and more consistent.

B.2 Tool Support

In this section we introduce the tool support of the approaches presented in Subsection 2.3.1. Namely, we identify the type of tool (internal, commercial, academic, ...) and the date of the latest release.

The tool support of dynamic approaches, compositional failure propagation approaches, and model-based transformational approaches are presented in Subsection B.2.1, Subsection B.2.2, and Subsection B.2.3 respectively.

B.2.1 Dynamic Approaches

Table B.2 displays the dynamic approaches addressed in Subsection 2.3.1 that have tool support for the specification and analysis of the dynamic behaviour of systems.

Table B.2: Tool-Support of the Dynamic Approaches

Approach - Work	Tool Support	Type of Tool	Latest Release
DFT - [Dugan92]	Galileo [Virginia03]	Commercial, Educational	2003
DFT - [Codetta-Raiteri05]	DrawNET (DFT), GreatSPN(GSPN)	Internal	2005
DFT - [Rao09]	DRSIM tool	Internal	2009
DFT - DFTCalc [Arnold13]	DFTCalc [Twente14]	Available	2014
DFT - Radyban [Montani08]	Radyban [Montani08]	Internal	2011
DFT - MatCarloRe [Manno12b]	MatCarloRe Tool [Manno14a]	Academic evaluation copy	2014
DFT - RAATSS [Manno14c]	RAATS Tool [Manno14b]	Academic evaluation copy	2014
RdP - [Signoret13]	BStoK [Workshop]	Comercial	2014
OpenSESAME - [Walter08]	OpenSESAME [Walter09]	Available	2009
BDMP - [Bouissou07]	KB3 Workbench [EDF14]	Available	2014
SEFT - [Kaiser07]	ESSaRel [Steiner12], TimeNET [TU Berlin07]	Internal*	2014

* Available for research purposes under agreement

B.2.2 Compositional Failure Propagation Approaches

Regarding the tool support of the Compositional Failure Propagation (CFP) approaches we can see that all approaches have been turned into tool-sets. Nonetheless, the CFP approaches are moving one step further, integrating dependability analysis models with design languages in order to link the design and analysis processes (cf. Subsection 2.3.1).

Table B.3: Tool-Support of the CFP Approaches

Approach - Work	Tool Support	Type of Tool	Latest Release
FPTN	SSAP Toolset [Fenelon93]	Unavailable	2006
HiP-HOPS [Papadopoulos11]	HiP-HOPS Tool [Hull14]	Available	2014
CFT	ESSaRel tool [TU Kaiserslautern09]	Available	2009
SCM [Domis09b]	ComposeR	Internal	2012
FPTC	Epsilon [Paige08b]	Available	2009
[Priesterjahn11a]	MechatronicUML, Fujaba [Paderborn12]	Available	2012

B.2.3 Transformational Approaches

As it is shown in Table B.4, all Architectural Design Languages (ADL) have their own implementation tool-sets. Namely, transformations from ADL models into compositional failure propagation models have been carried out through metamodels and profiles implemented as plugins.

Table B.4: Tool-Support of the Transformational Approaches

Approach - Work	Tool Support	Type of Tool	Latest Release
Simulink	Matlab [MathWorks14]	Comercial	2014
UML, SysML	e.g., Eclipse Papyrus [Eclipse12]	Available	2014
AltaRica	e.g., AltaRica Tools [Labri14]	Available	2014
AADL	e.g., Osate [CMU12]	Available	2014
CHESS-ML	CHESS Plugins [CHESS12]	Partially available	2012
FPTC	Epsilon [Paige08b]	Available	2009
Adler et al. [Adler10a]	CFT UML Profile	Internal	2012
HiP-HOPS	EAST-ADL2 Eclipse Plugin [ATESST10]	Available	2010
LARES [Ried112]	LARES toolset [Gouberman14]	Available	2014
Cressent et al. [Cressent11]	MÉDISIS Framework	Internal	2012

Analysis of Literature Approaches on a System Example

In this chapter we will focus on a hypothetical simple example to highlight the strengths and drawbacks of some of the approaches reviewed in Chapter 2.

This chapter is organised into the next sections:

- Section C.1 applies traditional Static Fault Trees [[Vesely02](#)] on the example system.
- Section C.2 uses Component Fault Trees [[Kaiser03](#)] on the example system.
- Section C.3 employs Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) approach [[Papadopoulos11](#)] on the example system.
- Section C.4 makes use of repairable Dynamic Fault Trees through the Reliability Availability Adaptive Transition System Solver (RAATSS) tool [[Manno14c](#)] on the dynamic example system.
- Section C.5 applies Structure Function of Dynamic Fault Trees [[Merle14](#)] on the dynamic example system.
- Section C.6 uses Boolean logic Driven Markov Processes (BDMP) [[Bouissou07](#)] on the dynamic example system.
- Section C.7 models the dynamic example system using State-Event Fault Trees (SEFT) [[Kaiser07](#)].

C.1 (Static) Fault Tree [Vesely02]

As Figure C.1 shows, the simultaneous failure occurrence of two subsystems (IE4, IE5) causes the system failure (IE1). These subsystems are characterized by the failure behaviour of their inner basic events (IE4: BE1, BE2, BE3; IE5: BE2, BE4, BE5). There exist other two combinations that also cause the system failure (IE2, IE3), which are characterized accordingly with their underlying basic events.

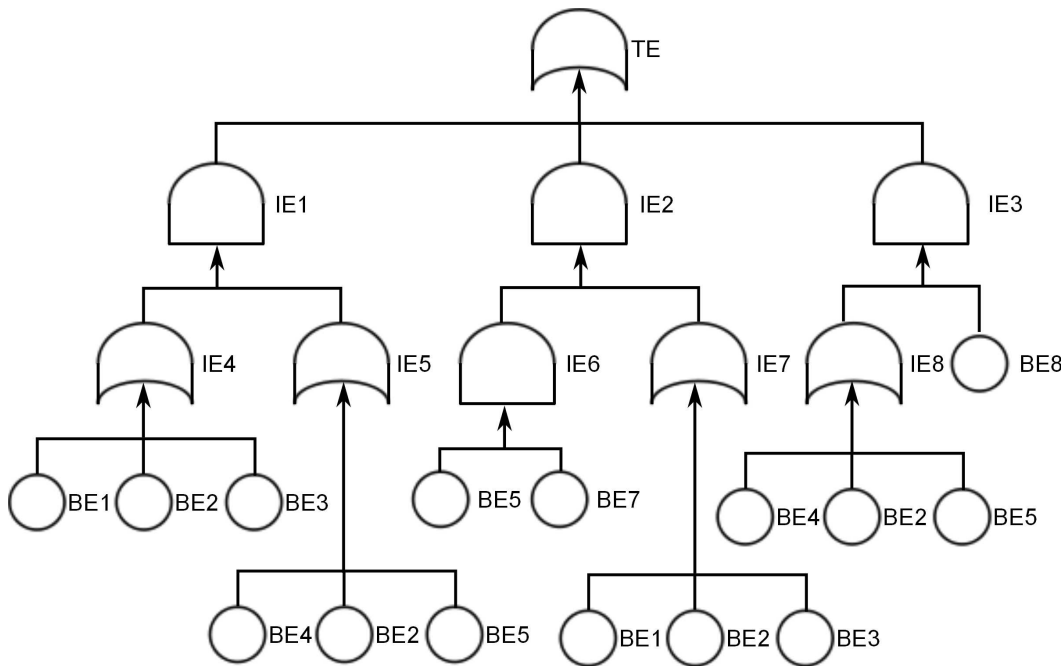


Figure C.1: Example System: (Static) Fault Tree Model

Note that this model contains repeated components/subsystems (IE4 ↔ IE7, IE5 ↔ IE8) and repeated basic events. In this example we left out the dynamic characteristics that the system's failure behaviour may contain, since this is one of the well-known drawbacks of static Fault Trees: inability to grasp dynamic characteristics of the system. Another obstacle worth considering is the flatness of the model. For complex systems the manageability, legibility and maintainability of the model becomes tedious and error-prone. However, due to the simplicity of the Fault Tree modelling process, still it is a widely used choice.

C.2 Component Fault Tree (ESSaReL tool) [Kaiser03]

To overcome the inability of static Fault trees to deal with complex systems, Component Fault Trees were introduced. In this simple example we have enclosed IE4 and IE5 components/subsystems and reused them to connect to the required gates across the model (see Figure C.2).

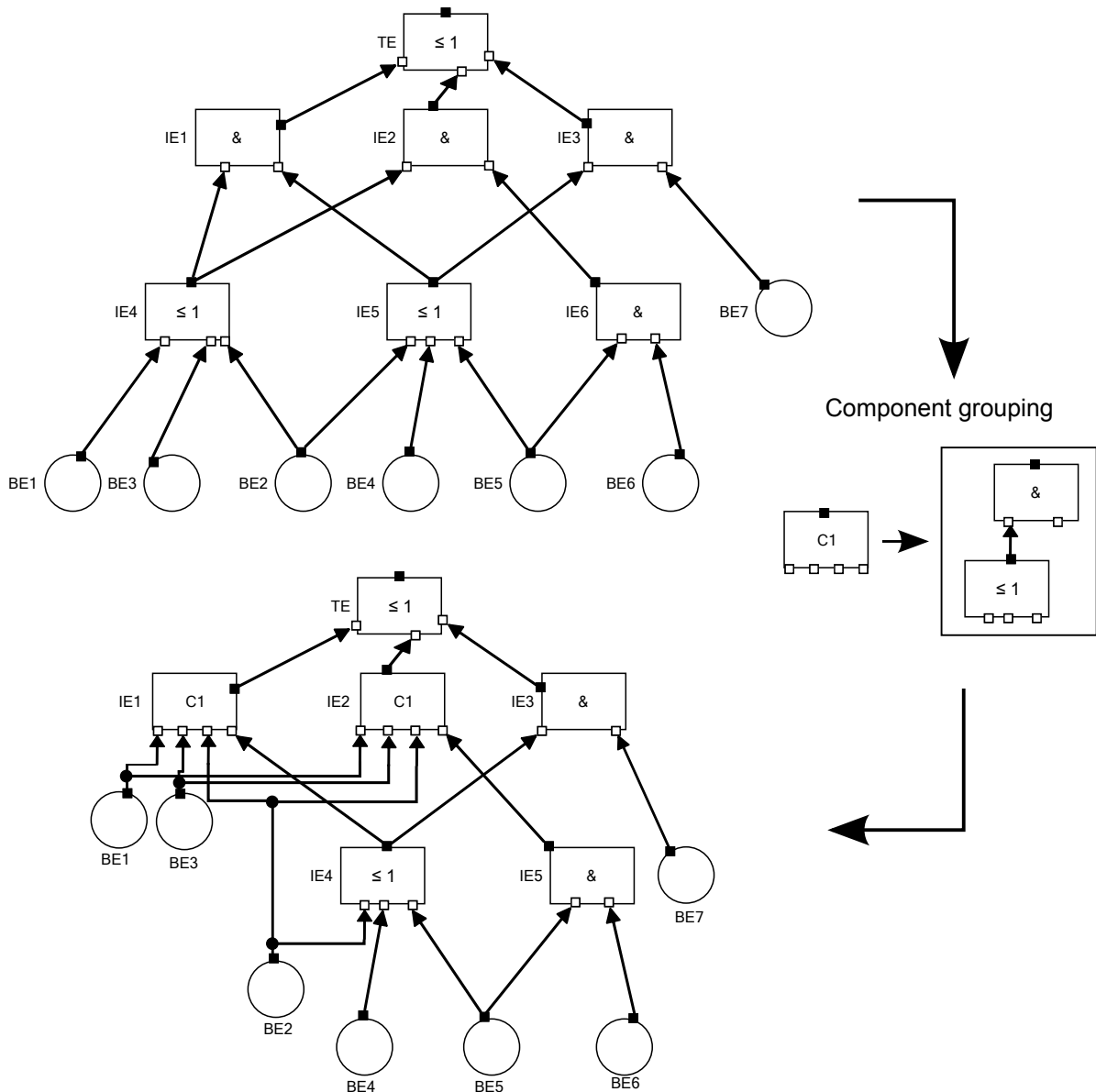


Figure C.2: Example System: Component Fault Tree Model

As Figure C.2 displays, the resulting model can be presented in a more intuitive manner than the traditional (static) Fault Tree model.

As with static Fault Trees, Component Fault Trees are unable to grasp the dynamic characteristics of the system. In their positive side, note that they are able to deal with repeated events and more importantly with repeated components, so that the readability and manageability of the whole model is improved.

C.3 HiP-HOPS [Papadopoulos11]

HiP-HOPS (Hierarchically Performed Hazard Origin and Propagation Studies) enables to deal with complex systems focusing on the component-based design concept.

Each design component is annotated with their corresponding failure behaviour and these components are connected to perform the system function. By propagating the failure annotations of each component, the static Fault Tree of the system is generated automatically. The whole the system can be seen as a forest of interconnected Fault Trees [Papadopoulos11]. Figure C.3 depicts the example system using HiP-HOPS annotations.

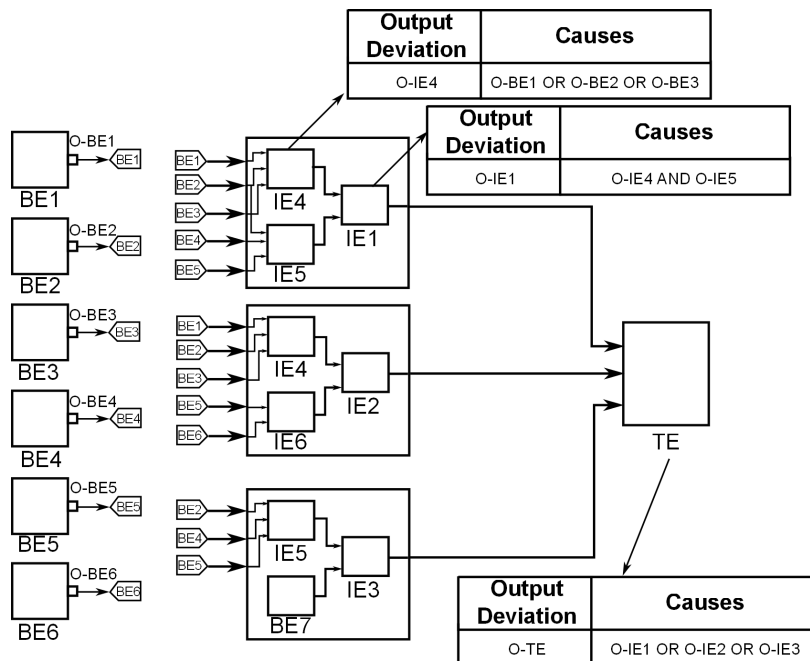


Figure C.3: Example System: HiP-HOPS Model

Apart from the automatic construction of Fault Tree models, HiP-HOPS is able to generate FMEA models and it implements automatic SIL decomposition and allocation techniques [Hull14].

Despite an extension to the dynamic failure characterization of HiP-HOPS have been done [Walker09], the approach is not able to grasp the dynamic characteristics of the system completely. As with Component Fault Trees, HiP-HOPS can deal with repeated events and repeated components.

C.4 Repairable Dynamic Fault Tree (RAATSS tool) [Manno14c]

In order to refine the system's failure behaviour, let us assume that some failure occurrences are required to occur sequentially: IE6 have to occur prior to IE7 and IE8 have to occur prior to BE8. Previously described models are unable to capture this logic, but the Dynamic Fault Tree (DFT) approach has the Priority AND (PAND) gate, which addresses this logic adequately. To analyse the system using the repairable DFT approach we will focus on the RAATSS tool (see Figure C.4).

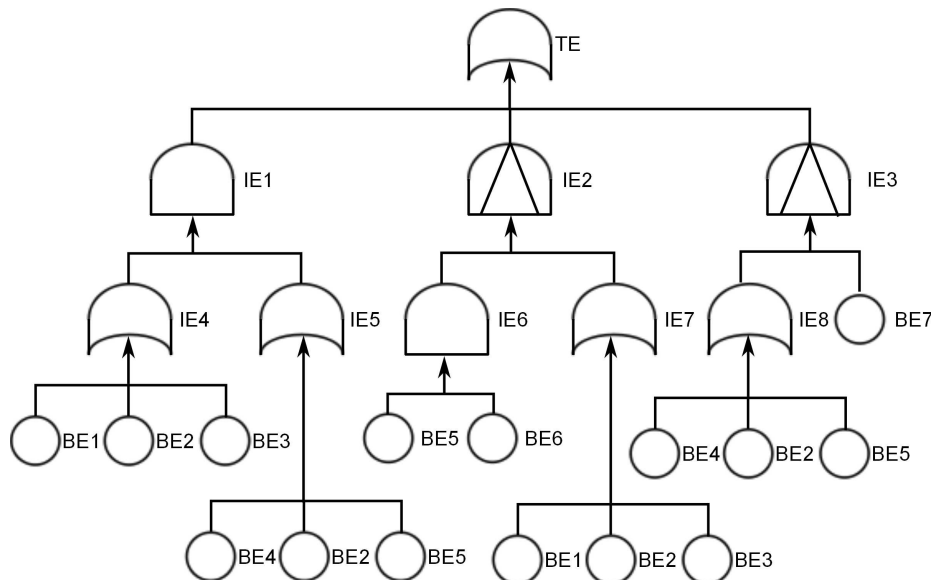


Figure C.4: Dynamic Example System: Dynamic Fault Tree Model

RAATSS enables the dynamic analysis of systems with repairable basic events. Besides, it makes possible modelling any failure/repair distributions. However, its main issues arises from the flatness of the model. As noted with static Fault Trees, large-flat models are difficult to maintain and understand.

C.5 Structure Function of Dynamic Fault Trees [Merle14]

The dynamic system example can be analysed by the algebraic framework for non-repairable Dynamic Fault Trees proposed by [Merle14].

Assuming that two events cannot happen at the exact same time we will characterize the example system of Figure C.4 as follows¹⁷:

$$\begin{aligned}
TE &= IE1 + IE2 + IE3 \\
IE1 &= (BE1 + BE2 + BE3).(BE4 + BE5 + BE2) = BE2 + (BE1 + BE3).(BE4 + BE5) \\
IE2 &= (BE5.BE6) \triangleleft (BE1 + BE2 + BE3) \\
IE2 &= (BE5) \triangleleft (BE1 + BE2 + BE3).(BE6) \triangleleft (BE1 + BE2 + BE3) \\
IE2 &= (BE5 \triangleleft BE1).(BE5 \triangleleft BE2).(BE5 \triangleleft BE3).(BE6 \triangleleft BE1).(BE6 \triangleleft BE2).(BE6 \triangleleft BE3) \\
IE3 &= (BE4 + BE2 + BE5) \triangleleft (BE7) \\
IE3 &= (BE4 \triangleleft BE7) + (BE2 \triangleleft BE7) + (BE5 \triangleleft BE7) \\
TE &= BE2 + BE1.BE4 + BE1.BE5 + BE3.BE4 + BE3.BE5 \\
&\quad + (BE5 \triangleleft BE1).(BE5 \triangleleft BE2).(BE5 \triangleleft BE3).(BE6 \triangleleft BE1).(BE6 \triangleleft BE2).(BE6 \triangleleft BE3) \\
&\quad + (BE4 \triangleleft BE7) + (BE2 \triangleleft BE7) + (BE5 \triangleleft BE7)
\end{aligned} \tag{C.1}$$

The canonical form of TE is the sum of all its Cut Sequence Sets (CSS) [Tang04]. In the compact form it is expressed as follows:

$$TE = \sum_{i=1}^n CSS_i \tag{C.2}$$

It is necessary to check for non-redundant CSS terms (denoted as \mathcal{S}_{min}) by applying

¹⁷Symbol \triangleleft denotes the before operator

the algorithm defined in [Merle10]. Assuming that there are m ($m \leq n$) non-redundant cut sequence sets, the probabilistic value of the TE can be calculated applying the inclusion-exclusion principle [Trivedi02]:

$$\begin{aligned}
Pr\{TE\} &= Pr\{CSS_1 + CSS_2 + \dots + CSS_M\} \\
&= \sum_{1 \leq i \leq m} Pr\{CSS_i\} \\
&\quad - \sum_{1 \leq i < j \leq m} Pr\{CSS_i.CSS_j\} \\
&\quad + \sum_{1 \leq i < j < k \leq m} Pr\{CSS_i.CSS_j.CSS_k\} + \dots + (-1)^{m-1} Pr\{CSS_1.CSS_2.CSS_m\}
\end{aligned} \tag{C.3}$$

with $\forall i \in 1, \dots, m, CSS_i \in \mathcal{S}_{min}$.

After verifying that there are no redundancies in the CSS terms of Equation C.1, we apply the inclusion-exclusion formula to the 9 independent cut sequence sets of Equation C.1. the resulting disjoint terms are 511¹⁸. Then the corresponding probabilistic formula should be applied to each term separately [Merle10]:

$$\begin{aligned}
Pr\{a.b\}(t) &= F_a(t) \times F_b(t) \\
Pr\{a + b\}(t) &= F_a(t) + F_b(t) + F_a(t) \times F_b(t) \\
Pr\{a \triangleleft b\}(t) &= \int_0^t f_a(u)(1 - F_b(u))du \\
Pr\{b(a \triangleleft b)\}(t) &= \int_0^t f_b(u)F_a(u)du
\end{aligned} \tag{C.4}$$

The algebraic framework proposed by Merle is adequate for small systems. However, when analysing real complex systems the process becomes tedious and prone to errors. Automated tool support to aid in the analysis process would improve its application.

C.6 BDMP [Bouissou07]

Boolean Driven Markov Process can be seen as a generalization of Dynamic Fault Trees [Bouissou07]. Such a generalization is achieved by the use of a trigger and triggered

¹⁸ $\sum_{i=1}^N \binom{i}{N}$ where $N=9$

Markov processes.

Figure C.5 depicts the BDMP model of the dynamic system example depicted in C.4. Although the BDMP approach enables connecting the output of a gate to the input of other multiple gates, the component-based concept is not integrated in the approach. That is, it is not possible to embed user-defined logic in a component and reuse it throughout the model. Refer to Subsection 2.3.1 to see other characteristics and limitations of the BDMP approach.

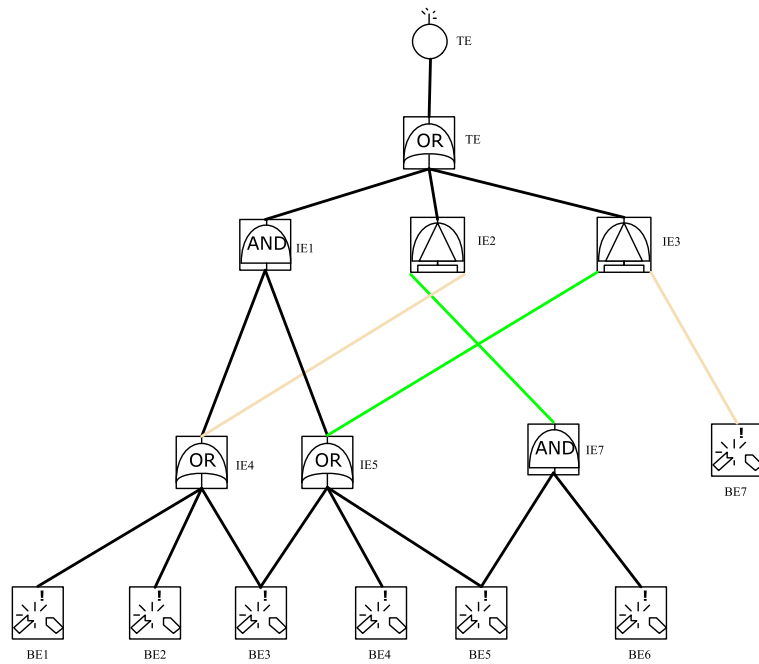


Figure C.5: Dynamic Example System: BDMP Model

C.7 SEFT - DSPN [Kaiser07]

State-Event Fault Trees (SEFTs) are able to analyse the system's failure and repair behaviour through the use of components. To this end, the approach models the system failure/repair behaviour using SEFT concepts and subsequently this model is transformed into the TimeNET tool [Ilmenau07] for the analysis of its corresponding Deterministic and Stochastic Petri Nets (DSPN) model. Therefore, the failure and repair occurrence of its events will be characterized according to exponential and deterministic transitions.

Basic SEFT modelling mechanisms include (cf. Figure C.6): (1) states (e.g., $BE1_OK$), (2) transitions (e.g., $fault$) and (3) state/event ports (e.g., $BE1_F$). Besides, different gates are modelled in SEFT formalism: Priority AND (P&), OR (≥ 1), NOT and more (see [Kaiser07] for a more detailed definition of all the gates). All these modelling mechanisms have defined their own counterpart in the DSPN modelling: both formalisms include transitions, SEFT states are modelled through DSPN places and each SEFT gate has associated its corresponding DSPN net. Thus, so as to analyse an SEFT model its transformation to DSPN model is necessary.

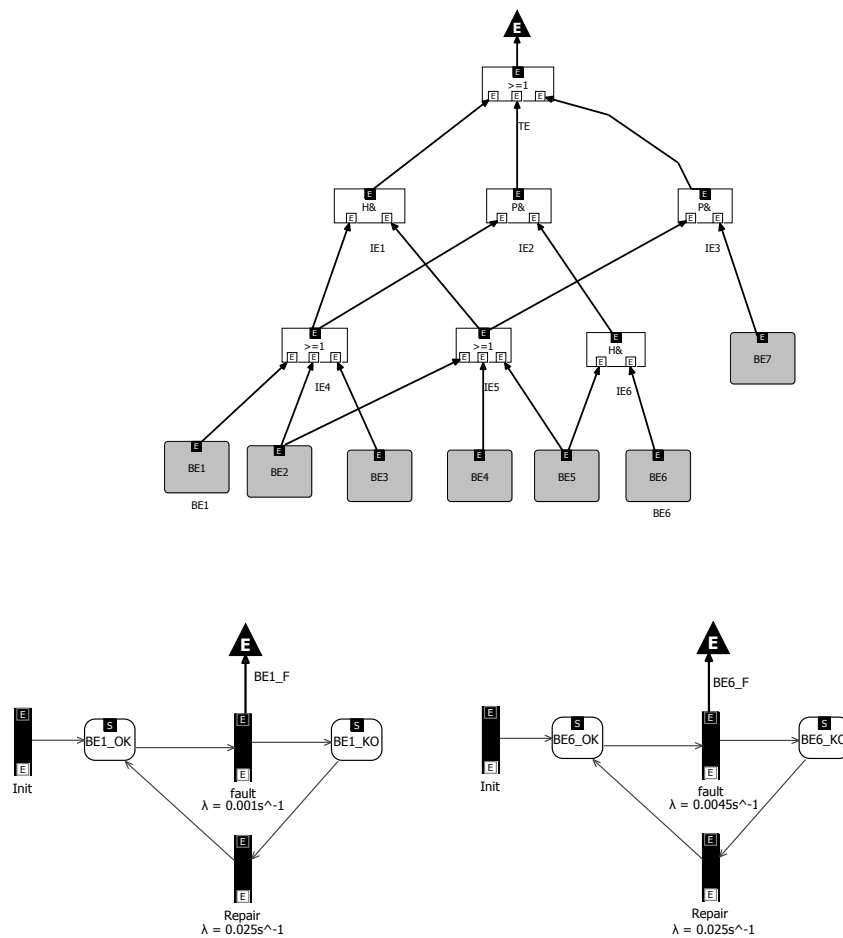


Figure C.6: Dynamic Example System: SEFT Model

Once the SEFT model of the dynamic example system shown in Figure C.4 is created (see Figure C.6), its transformation results in the DSPN model depicted in Figure C.7. As it can be seen from the DSPN model (cf. Figure C.7) the resulting dependability

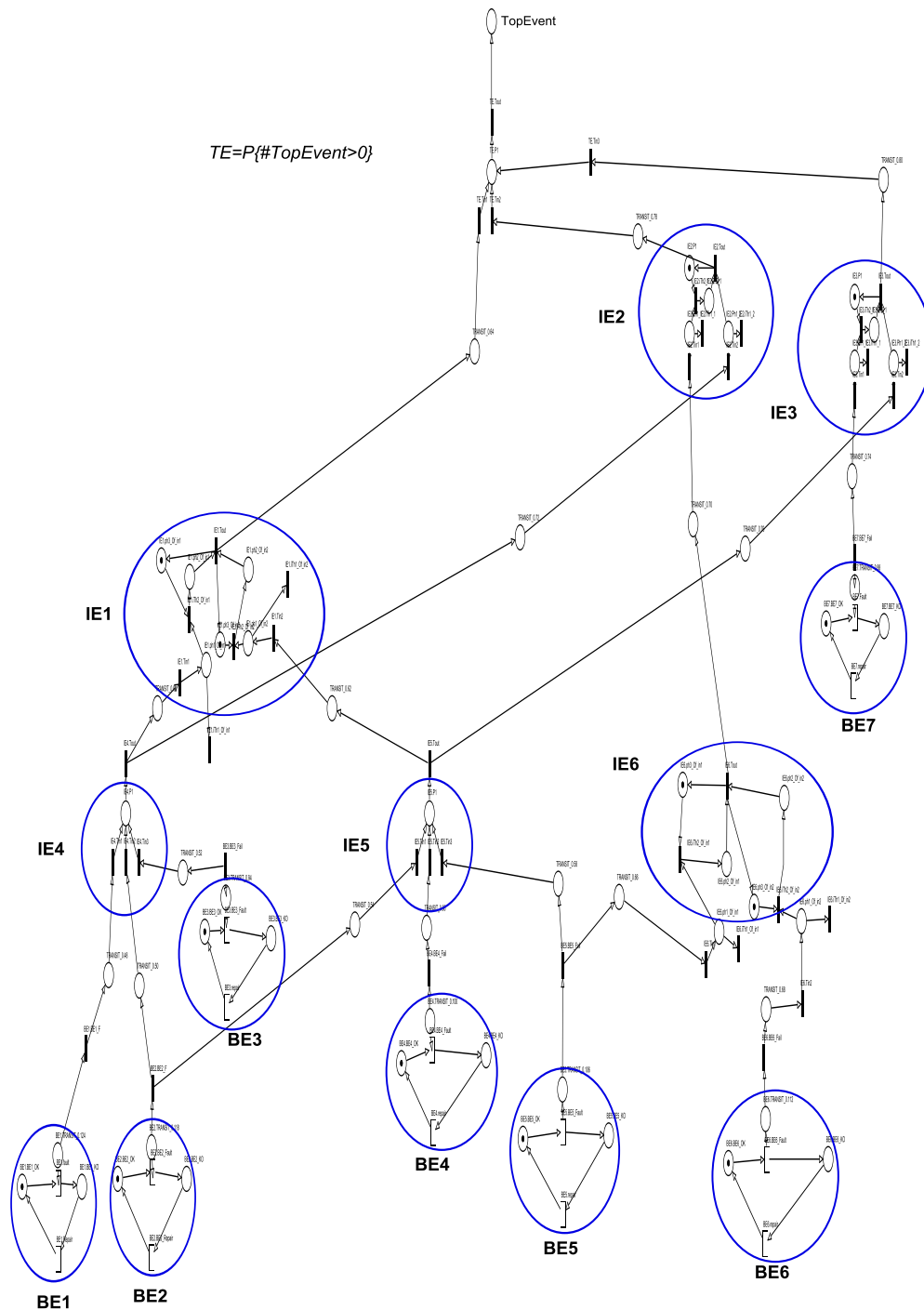


Figure C.7: Dynamic Example System: SEFT's Underlying DSPN Model

analysis model is a flat DSPN model. Therefore, for complex systems, tracing from the SEFT model towards the DSPN model is not straightforward and it can suffer from the state-explosion problem. Another issue worth considering is the fact that it

is not possible to connect CFTs with SEFT models in order to trigger SEFT model's state changes through Component Fault Tree's top-events. These characteristic would make the approach even more expressive and open the way to analyse further complex systems. In its actual version it manages effectively repairable basic events and it is able to include functional-design information through component-based characterization.

Automation/Implementation of the HW/SW Architecture Design

To implement and automate the construction of the *extended HW/SW architecture* a model-based design approach has been implemented [Aizpurua13a]. As described in the Figure D.1 the design process is specified as follows:

- (1) The process starts from the construction of the system architecture model specified in Simulink. System's resources (sensors, controllers, actuators, network) are modelled using Simulink's subsystem blocks. At the highest or top level, the system architecture model is characterized as a set of connected subsystem blocks, which will have internally their corresponding functionality and logic.
- (2) System's implementations (which will be comprised of resources) are characterized based on the token-based specification (cf. Characterization 3.1) according to the Functional Modelling Approach (FMA) and Extended Functional Modelling Approach (EFMA). To this end, Simulink model's subsystem blocks are annotated by previously defined token-based specifications. Thus, each subsystem (resource) block in the model will have its own description annotated in a underlying *xml* character string with the predefined fields specified according to the FMA and EFMA (see Subsection 3.3.1 and Subsection 3.3.4).
- (3) Once all the system resources are annotated with their characteristics, the underlying *xml* annotations of the model's blocks are processed. Thanks to the algorithm defined for the identification of heterogeneous redundancies (see Algorithm 2 and Algorithm 3), the approach automatically suggests a list of possible heterogeneous redundancies.

- (4) Finally, after processing the annotations of system resources, the reconfiguration table is extracted. The reconfiguration table identifies all the possible system implementations to perform the main function. These implementations are ordered according to their implementation priority.

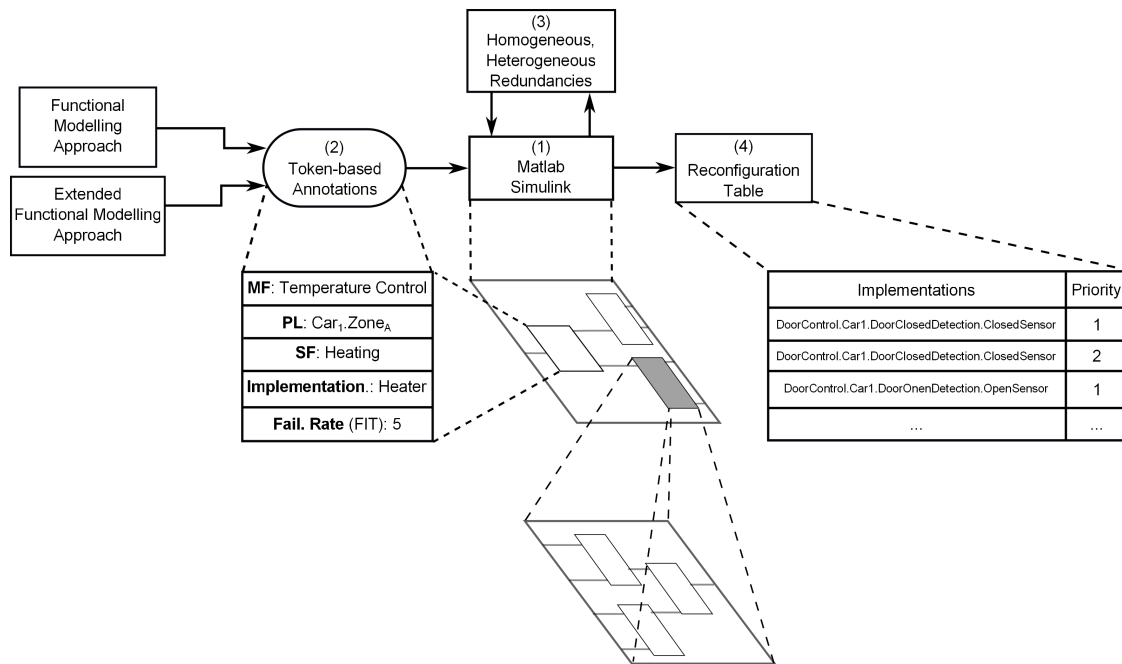


Figure D.1: D3H2 Metodology: Design Implementation

D.1 Annotations of the System Architecture

The annotations of the Simulink model's subsystem blocks are characterized with predefined data fields for exploitation purposes. Two main data structures have been designed to characterize each system implementation: (1) functional; and (2) failure data structures.

As for the first structure, depending on the implementations' subfunction type (I, C, O), we further divide the functional data structure into two main groups: (1) input and output resource implementations and (2) control resource implementations. While input and output resource implementations enclose the corresponding logic in a single block, control resources (i.e., processing units) has further inner subsystems blocks in

order to model the allocated SW tasks: control, fault detection, reconfiguration and fault detection of the reconfiguration SW implementations.

Depending on the type of resource implementation, we define mandatory and optional fields. Initially, the subsystem blocks will not have information about the redundancies that may exist in the system model. Therefore, this information field is not necessary when annotating subsystem blocks, and likewise, the information concerning the priority of the implementation is not necessary in all the cases. Hence, each subsystem block has a functional data structure with the following fields:

- Main Function (MF): mandatory field.
- Subfunction Type (SFC): I, C, O, FD, R or FD_R. Mandatory field.
- Subfunction (SF): mandatory field.
- Physical Location (PL): mandatory field.
- Implementation: mandatory field.
- Priority: optional field for redundant implementations (priority > 1); otherwise (priority=1) mandatory.
- Allocated: mandatory for PUs. Each PU may have allocated (many) different implementations. This field encloses all its inner SW resources, where each resource has the next mandatory fields: (1) Identifier, (2) MF, (3) SFC, (4) SF, (5) PL, (6) Implementation, (7) Priority.
- Redundancy: this data field is further divided into homogeneous and heterogeneous fields. Once the system model is analysed to search possible redundancies, possible candidates are classified as homogeneous or heterogeneous redundancy and they are annotated to the corresponding resource as a potential redundancy for the implementation at hand. Each candidate redundancy implementation has the next fields: MF, SFC, SF, PL, Implementation, Priority, and Full Name (or identifier).

The token (field) of the physical location may contain different detail (depth) levels depending on the physical location. The token of the physical location is stored in a variable as an array of strings with its dimension equal to the depth of the physical

location. For instance, to store the annotation: $PL=Car_1.Zone_A$, we will use an array of length 2.

As for the failure data annotations, the next set of data fields have been defined as mandatory for all the implementation resources:

- Implementation: name of the implementation.
- Description: character string defining the implementations purpose.
- Distribution: probabilistic failure distribution of the implementation: exponential, constant, or Weibull.
- Failure Rate (MTTF) and shape parameter (Weibull).
- Time Unit: Billion Hours, Years, Weeks, etc.
- Cost: monetary cost of the implementation (Euro).

To aid the designer when filling the necessary fields, these data is loaded from a existing database automatically. This is achieved by identifying the (possible) matching implementation's name (or identifier) in the database and accordingly, suggesting all possible implementation names, failure rates, and cost values.

D.2 Identification of Heterogeneous Redundancies

In this subsection we will introduce the algorithms for the identification of heterogeneous redundancies. To this end, we focus on a Simulink model constituted of different blocks each of them annotated with functional data structure fields.

The algorithm for the identification of heterogeneous redundancies arising from natural compatibilities takes as input (Algorithm 2):

- $BLOCKS_{sys}$: An array of strings with its dimension equal to the number of implementation (subsystem) blocks in the model. Each element of the array characterizes functional annotations of the implementation and accordingly, the whole set of strings specifies the design annotations of the system.

- *BLOCK_{check}*: The implementation to be checked in order to find homogeneous or heterogeneous redundancies.

Besides, the algorithm makes use of the next functions:

- (1) $Y = xml_parse(A)$ function (line 2, 15) with the next input and output variables respectively: A : an *xml* character string (in our case it will specify the underlying *xml* of the system model); Y : the data structure corresponding to the *xml* string, with the attributes of the data fields in the *xml* string (accessible as $Y.attribute$).
- (2) $Y = get_SubSystemComponents(A)$ function (line 6, 19) with the following input and output variables respectively: A : a variable specifying the reference to a block; Y : an array of *xml* strings with the corresponding underlying annotations of each inner block in A .

Basically, the algorithm for the identification of heterogeneous redundancies arising from natural compatibilities checks whether same subfunctions are located in contiguous physical locations (cf. line 34).

This algorithm does not provide any output, instead it updates directly the Simulink model through the *AnnotateRedundancy(A, B, C)* function (see line 39). The input parameters of the *AnnotateRedundancy* function are:

- A : Block of the Simulink model to be annotated.
- B : An *xml* character string specifying the redundancy annotations (see Section D.1).
- C : Type of redundancy (homogeneous or heterogeneous).

The identification of heterogeneous redundancies arising from forced compatibilities is not an automatically performed task. However, Algorithm 3 outlines the process to create a list with possible heterogeneous redundancy suggestions. It is the designer who should have to analyse this list thoroughly to check if it is possible to use any of the suggested implementations with additional resources as an heterogeneous redundancy for the indicated subfunction.

The outlined algorithm for the identification of heterogeneous redundancies arising from forced compatibilities (Algorithm 3) takes as input variables the same data variables as

Algorithm 2 Heterogeneous Redundancy Identification (Natural Compatibility)

```
1: function HeteRedIdentification( $BLOCKS_{sys}$ ,  $BLOCK_{check}$ )
2:    $xml_a = xml\_parse(BLOCK_{check})$ ; // parse data structure
3:    $allocated_a = 0$ ; // variable indicating if the implementation is a PU
4:   if ( $strcmp(xml_a.SFC, 'C')$ ) then // check subfunction component
5:      $allocated_a = 1$ ; // indicate that it is a PU
6:      $BlockList_a = get\_SubsystemComponents(block_c)$ ; // get SW implementations
7:   for  $j = 1 : |xml_a|$  do
8:     if ( $allocated_a$ ) then // if SFC='C' get inner data
9:        $xml_a = BlockList_a(j)$ ; // get SW implementations of the PU
10:     $Hw_A = xml_a.name$ ; // name
11:     $SF_A = xml_a.SF$ ; // subfunction
12:     $I_A = xml_a.Implementation$ ; // identifier
13:    if ( $xml_a.Priority$ ) then // nominal implementation? (Priority=1)
14:      for  $k = 1 : |BLOCKS\_SYS|$  do // parse all the system blocks one by one to
// find heterogeneous redundancies for the nominal implementation in  $xml_a$ 
15:         $xml_b = xml\_parse(BLOCKS\_SYS(k))$ ;
16:         $allocated_b = 0$ ; // variable indicating if the implementation is a PU
17:        if ( $strcmp(xml_b.SFC, 'C')$ ) then // is it a control implementation?
18:           $allocated_b = 1$ ; // indicate that it is a PU
19:           $BlockList_b = get\_SubsystemComponents(xml_b)$ ;
20:        for  $i = 1 : |xml_b|$  do
21:          if ( $allocated_b$ ) then
22:             $xml_b = BlockList_b(i)$ ;
23:             $Hw_B = xml_b.name$ ; // name
24:             $SF_B = xml_b.SF$ ; // subfunction
25:             $I_B = xml_b.Implementation$ ; // identifier
26:             $type = ''$ ; // homogeneous or heterogeneous redundancy
27:            if ( $strcmp(SF_A, SF_B)$ ) then // if SFs match
28:              if ( $any(strcmp(xml_a.SFC, \{ 'I', 'O' \}))$ ) then
29:                 $P_A = xml_a.PL$ ;
30:                 $P_B = xml_b.PL$ ;
31:                 $sameHw = strcmp(Hw_a, Hw_b)$ ; // same Simulink block?
32:                if ( $strcmp(P_a(:), P_b(:))$ ) AND ( $\sim sameHw$ ) then
33:                   $type = 'homogeneous'$ ; // same exact PL
34:                else if ( $(|P_a| == |P_b| == 2)$ ) AND  $strcmp(P_a(1), P_b(1))$  then
35:                   $type = 'heterogeneous'$ ; // same car, different zone
36:                else if ( $sameHw$  AND  $\sim strcmp(I_A, I_B)$ ) OR  $\sim sameHw$  then
37:                   $type = 'homogeneous'$ ;
38:                if ( $\sim isempty(type)$ ) then
39:                   $AnnotateRedundancy(BLOCK_{check}, xml_b, type)$ ;
```

the Algorithm 2: $BLOCKS_{sys}$ and $BLOCK_{check}$.

Besides it makes use of an additional function $checkPhysicalCompatibility(A, B)$ which evaluates if the physical location of the implementations are compatible or not. The algorithm (or rules) to evaluate possible compatible physical locations has been outlined in Subsection 3.3.2. This function was not used with the Algorithm 2 because depending on the specific case of the physical compatibility, the algorithm determines if redundancies are homogeneous or heterogeneous (see Algorithm 2 lines [32-37]).

It does not provide any output variable, instead it updates the Simulink model and its underlying annotations directly calling the function $AnnotateSuggestion(A, B)$ (line 14) where its input parameters are:

- A : Block of the model to be annotated
- B : Possible redundancy annotations.

Algorithm 3 Heterogeneous Redundancy Suggestion (Forced Compatibility)

```

1: function HeteRedSuggestions( $BLOCKS_{sys}, BLOCK_{check}$ )
2:    $xml_a = xml\_parse(BLOCK_{check});$ 
3:   if ( $any(strcmp(xml_a.SFC, \{ 'I', 'O' \}))$ ) AND ( $xml_a.Priority == 1$ ) then
4:      $MF_A = xml_a.MF;$  // Main Function
5:      $SFC_A = xml_a.SFC;$  //  $SFC = \{I, C, O\}$ 
6:     for  $k = 1 : |BLOCKS\_SYS|$  do // parse all the system blocks
7:        $xml_b = xml\_parse(BLOCKS\_SYS(k));$ 
8:       if  $\sim strcmp(xml_b.SFC, 'C')$  then // non-control implementations
9:          $MF_B = xml_b.MF;$ 
10:         $SFC_B = xml_b.SFC;$ 
11:         $compatiblePL = CheckPLCompatibility(xml_a.PL, xml_b.PL);$ 
12:        if ( $strcmp(SFC_A, SFC_B)$ ) AND
13:        ( $\sim strcmp(MF_A, MF_B)$ ) AND ( $compatiblePL$ ) then
14:           $AnnotateSuggestion(BLOCK_{check}, xml_b);$ 

```

D.3 Extraction of the Reconfiguration Table

Once all system implementations/components has been annotated with their characteristics, in order to extract the reconfiguration table it is enough to parse the xml string

of the model and extract each components annotations with their corresponding fields.

For the identified homogeneous and/or heterogeneous redundancies and for the implementations which do not have user defined priority, the prioritization of the implementations which constitute the reconfiguration table is based on:

- (1) Type of redundancy: we assume that homogeneous redundancies have higher priority than heterogeneous redundancies.
- (2) Physical distance between redundancies: among heterogeneous redundancies originating from natural compatibilities we set higher priority for those implementations which are closer to the nominal implementation. To this end, each Simulink model has its own physical location map. This map links qualitative physical location identification tokens, e.g., $\text{Car}_1.\text{Zone}_A$, with their corresponding quantitative space/plane coordinates as depicted in Figure 3.3.
- (3) Unreliability of the implementation.
- (4) Cost of the implementation.

Among equally weighted implementations, we focus on the weighted sum of the unreliability and cost to determine which implementation's priority is higher. As for the fault detection and reconfiguration implementations the designer should assign priorities to the respective implementations because these depend on design-specific assumptions. Concerning the fault detection of the reconfiguration implementation (FD_R), all these implementations have priority=1 because they operate as heartbeat (keepalive) implementations.

Algorithm 4 Reconfiguration Table Extraction Algorithm

```
1: function ReconfigurationTable = MAIN(BLOCK_SYS)
2:   SF_list = {}; // different SFs list, no repetitions
3:   pos = 0;
4:   for (i := 1 to |BLOCKS_SYS|) do
5:     xmla = xml_parse(BLOCKS_SYS(i)); // parse all the system blocks
6:     // no FD, R or FD_R AND (list is empty OR current SF is not already in the list)
7:     if ( $\sim$ any(strcmp(xmla.SFC, {FD, R, FD_R}))) AND
8:     ((isempty(SF_list) OR ( $\sim$ any(strcmp(SF_list{:}, xmla.SF)))) then
9:       dim = dim + 1;
10:      SF_list{dim} = xmla.SF; Add SF to the list
11:      groupSF = {}; // all implementations of the same SF
12:      x = 0;
13:      for (j := 1 to |BLOCKS_SYS|) do // parse system blocks to find matching
14:        xmlb = xml_parse(BLOCKS_SYS(j)); // parse all the system blocks
15:        if strcmp(SF_list{dim}, xmlb.SF) then // SF already exists in the list
16:          x = x + 1;
17:          groupSF{x} = xmlb; // group all the implementations of the same SF
18:          if x > 1 then // order implementations wrt priority
19:            groupSF = AssignPriority(groupSF);
20:          pos = pos + 1;
21:          table{pos} = groupSF; // store I, C, O implementations in the table variable
22:          pos = pos + 1;
23:          table{pos} = xmla; // store FD, R, FD_R implementations in the table variable
24:      ReconfigurationTable = createReconfigTable(table); // map the variable to the
25:      table
25: return ReconfigurationTable;
```

Failure Rate & Cost Data

The goal of this chapter is to present the failure rate, repair rate, and cost values of the different hardware, software, and communication resources.

Table E.1 displays the failure rate and cost values of the different hardware, software and communication resources. Despite the applied dependability analysis formalisms are independent from the statistical distribution of the failure and repair process (i.e., Component Dynamic Fault Trees and Stochastic Activity Networks), for the sake of simplicity in all the calculations exponential distributions have been assumed.

Table E.1: Failure Rates & Cost Values of HW, SW and Communication Resources

<i>Resource</i>	λ (<i>year⁻¹</i>)	μ (<i>year⁻¹</i>)	<i>Cost</i> (€)
SW_Det, SW_HM	1 E-2	0.5	80 each
SW_FP	1 E-2	0.5	-
Fire Detector [SINTEF09] + Mounting	3.77 E-2	0.5	20 + 60€/hour
Temperature Sensor [IAEA88]	1.49 E-2	0.5	-
Pressure Sensor [IAEA88] + Mounting	1.6 E-2	0.5	20 + 60€/hour
Speed Sensor + Mounting	1.8 E-2	0.5	20 + 60€/hour
Camera[jvc]	9.43 E-2	0.5	-
PU [Vinod08]	3.87 E-2	0.5	30
Comm. & Gateway	5 E-3	0.5	200

In Table E.1, resources with the same characteristics have been grouped as follows:

- *Pressure sensor* covers open, closed and obstacle detection sensors.
- *Processing unit* gathers the characteristics of all different PUs.
- *Communications* include MVB and Ethernet communication protocols and their gateway.

Regarding SW components, hypothetical reasonable values are assumed. As noted in previous chapters (see Section 4.5), the cost of SW components is quantified assuming that their development cost will be paid in 4 years.

We differ 4 type of SW components: (1) fault detection software (SW_FD); (2) re-configuration software (SW_R); (3) fault detection of the reconfiguration software (SW_FD_R); and (4) control/detection software (SW_Det).

The development costs of each of these 4 SW components is considered once for different implementations of the same subfunction: once developed they are adapted for the related implementations. This assumption is adopted because the grouped subfunction implementations are closely related and they do not need a significant development cost.

- All fault detection implementations (SW_FD) adapt to different subfunctions modifying subfunction-specific time/value thresholds.
- Reconfiguration implementations' development cost (SW_R) does not differ for different subfunctions, alternative implementations will have allocated different reconfiguration tables for different subfunctions, but the reactivation logic holds the same for different subfunction's reconfiguration implementations.
- Reconfiguration's fault detection implementations development cost (SW_FD_R) for different subfunctions differ only in the keepalive timeout, but their development is independent of the subfunction.
- All the considered control/detector software implementations (SW_Det) have a closely related logic, for instance, for the Door Status Control main function all detection implementations are linked with the position of the door.

SW_FD, SW_R, and SW_FD_R are gathered in the component *SW_HM* referring to the failure rate and cost values of health management software implementations. Each implementation of the *SW_Det* resource covers: *SW_OpenDetection*, *SW_ClosedDetection*, *SW_ObstacleDetection*, *SW_DoorVelocity*, *SW_DoorControl*, *SW_FireDetection* and *SW_FireControl* functionalities. Each of them is characterized with the same failure rate, repair rate and cost values. Accordingly, for the characterization of the false positive events (SW_FP) we have applied the same values as for the other software implementations.

The same repair rate values have been assumed for all the hardware, software and

communication resources.

With respect to the sensor's cost, human cost related with mounting and testing tasks is considered assuming 10 minutes/sensor at a rate of 60 €/hour.

Finally, note that the cost of some hardware resources have been excluded deliberately in Table E.1. The rationale under this decision is that they are used as heterogeneous redundancies. Therefore, they already exist in the system and they are not explicitly added to provide fail-over capabilities. This is why their use does not incur an increase in the hardware cost.

PAND Model for Repairable Systems

The goal of this chapter is to explain the behaviour, implementation and validation of the PAND gate's model for repairable systems used in Chapter 5.

When considering repairable systems, we assume that components can fail and repair repeatedly during the mission time of the system. The basic behaviour of the PAND gate model for repairable systems is as follows: it will trigger when the occurrence of events respect the sequence determined by the gate, e.g., assuming $Y = PAND(A, B)$; $Y = 1$ if A occurs prior to B and then B occurs; otherwise $Y = 0$. However, there are some details worth mentioning.

Our PAND gate model for repairable systems will consider the last failure of each of its input component instead of considering only their first failure - as it is done with non-repairable components. Furthermore, the restoration of the PAND gate for repairable systems (from $Y = 1$ to $Y = 0$) will be performed once that one of its input components is restored. The logic implemented in the PAND gate for repairable systems of this dissertation agrees with the logic implemented in the RAATSS tool [Manno14c].

For the implementation of the gate using the SAN formalism we consider 2 interconnected components (see Figure F.1): (1) the component A_BF_B checks whether the event A happens before the event B ; and (2) the component A_PAND_B checks that the order is respected (A before B) and that the event B occurs. If the order is not respected or B does not happen, the $PAND$ output will not happen as well. Note that the implemented PAND gate is not inclusive, i.e., simultaneous failure occurrences are not included because they don't respect the sequence.

The model A_BF_B characterizes the situation in which the event A fails prior to the event B (cf. Figure F.2). This event is defined through the input gates BF and no_BF

$$Y = \text{PAND}(A, B)$$

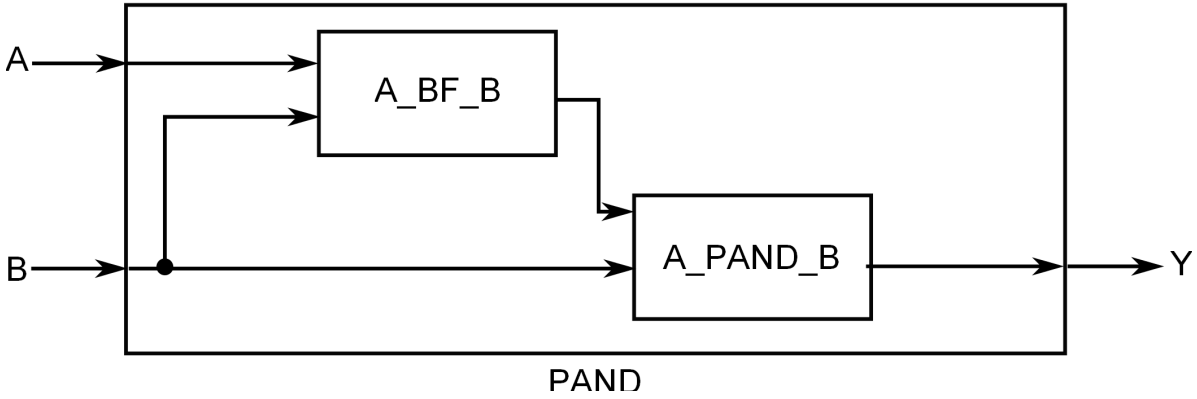


Figure F.1: Block Diagram of the Repairable PAND Model

respectively (cf. Table F.1).

Table F.1: Activities in the SAN model A_BF_B

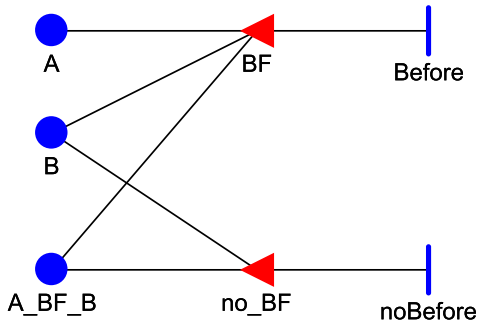


Figure F.2: Atomic Model of the Component A_BF_B

Input Gate:	BF
Input Gate Predicate:	$(m(A)=1 \ \&\& \ m(B)=0 \ \&\& \ m(A_BF_B)=0)$
Input Gate Function:	$m(A_BF_B)=1;$
Input Gate:	no_BF
Input Gate Predicate:	$(m(A_BF_B)=1 \ \&\& \ m(A)=0)$
Input Gate Function:	$m(A_BF_B)=0;$

The second model (A_PAND_B) characterizes the situation in which the event A have already failed prior to the event B (A_BF_B) and then the event B occurs (cf. Figure F.3).

As for the validation of the gate we have compared the results for different tests. For simplicity here we show only the basic configuration in which we have 2 basic events as inputs (A and B) with exponential failure rates of 0.1 and 0.3 respectively and exponential repair rate of 0.5 for both basic events. To this end, we have shared the

Table F.2: Activities in the SAN model A_PAND_B

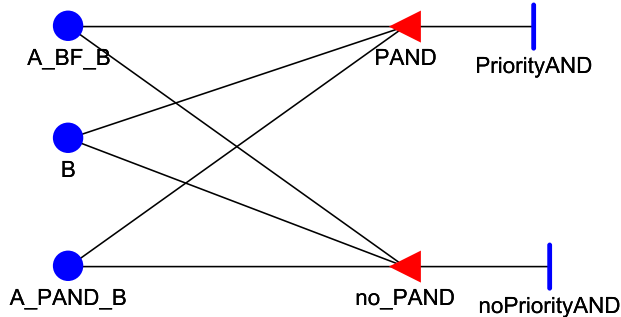


Figure F.3: Atomic Model of the Component A_PAND_B

Input Gate:	PAND
Input Gate Predicate:	$(m(A_BF_B)==1 \ \&\& \ m(B)==1 \ \&\& \ m(A_PAND_B)==0)$
Input Gate Function:	$m(A_PAND_B)=1;$
Input Gate:	no_PAND
Input Gate Predicate:	$(m(A_PAND_B)==1 \ \&\& \ (m(A_BF_B)==0 \ \ m(B)==0))$
Input Gate Function:	$m(A_PAND_B)=0;$

places of the basic events' failed places (characterized as in Figure 5.3) with the events A and B characterized in this Chapter. Figure F.4 displays the output obtained using SAN and ATS formalisms using the Mobius and RAATSS tools respectively.

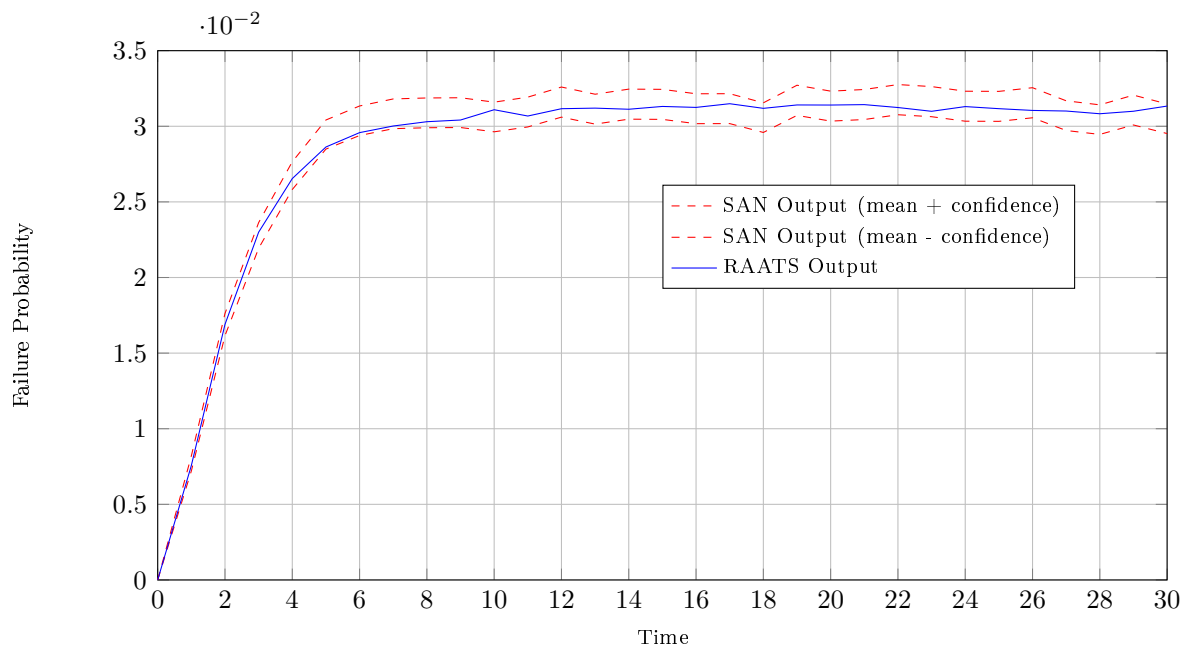


Figure F.4: Repairable PAND gate using Mobius and RAATSS Tools

List of Figures

1.1	Massively Networked Scenario: Railway Train Example	4
1.2	Internal Architecture of a Building: Functions and Communication Interfaces	5
2.1	Train Car Configuration: Functions and Communication Interfaces	12
2.2	Train Car Configuration: Physical Distribution	13
2.3	Hardware Model of the Door Status Control Function	14
2.4	SW/Dependency Model of the Door Status Control Function	14
2.5	Hardware Model of the Video Surveillance Function	15
2.6	SW/Dependency Model of the Video Surveillance Function	15
2.7	Hardware Model of Air Condition Control Function	16
2.8	SW/Dependency Model of Air Condition Control Function	16
2.9	Hardware Model of the Fire Protection Function	16
2.10	SW/Dependency Model of the Fire Protection Function	16
2.11	Hardware Model of the Passenger Information System	17
2.12	SW/Dependency Model of the Passenger Information System	17
2.13	Hardware Model of the Light Control Function	18
2.14	SW/Dependency Model of the Light Control Function	18
2.15	Triple Modular Redundancy Example	26
2.16	Diverse Design [Littlewood00a]	27
2.17	Dynamic Fault Tree Symbols	35
2.18	Dynamic Parametric Fault Tree Example	37
2.19	Composition Aggregation Method of [Arnold13]	37
2.20	Hierarchical Structure and CFP Annotations in HiP-HOPS	42
2.21	Methodology for Designing Distributed Control Systems [Cauffriez04]	50
2.22	Design Approach of [Clarhaut09]	51

2.23	Example of an Adaptation Specification View [Adler10b]	55
3.1	D3H2 Methodology [Aizpurua13a]	67
3.2	Functional Modelling Approach	69
3.3	Example of a Train Physical Location Map	70
3.4	Functional Modelling Approach for Existing Systems	72
3.5	Abstract Architecture of the Main Function i and the Health Management Implementation of its Output Subfunction	79
4.1	Extended HW/SW Architecture's Failure Model	103
4.2	Component Dynamic Fault Tree Overview	110
4.3	Component Dynamic Fault Tree Example	113
4.4	FCI Example Time-Diagrams	117
4.5	Failure Criticality Index Calculation Example	119
4.6	Overview of the Uncertainty Analysis	121
4.7	Relative Failure Probability & Cost of Fire Protection Control Configurations (10^6 iterations)	128
4.8	Failure Probability of Fire Protection Control Configurations under Different Assumptions (10^6 iterations)	131
4.9	Fire Protection Control Failure Probability Distribution: Communication's Failure Rate Influence ($10^4 \times 5.10^3$ iterations)	133
4.10	Relative Failure Probability & Cost of Alternative Door Status Control Main Function's Configurations for the Train.Car ₁ .Zone _A .Door (10^6 iterations)	139
4.11	FCT_{DOD} - Heterogeneous Redundancy (10^6 iterations)	140
4.12	FCT_{DOD} - Homogeneous Redundancy (10^6 iterations)	140
4.13	Door Status Control: Ideal Configurations Relative Failure Probabilities w.r.t. Reference Configuration (10^6 iterations)	142
4.14	Failure Probability Distribution: 2R Centralised Heterogeneous Configuration - Communication's Failure Rate Intervals ($10^4 \times 5.10^3$ iterations)	145
5.1	Challenges Emerging from Repairable Systems (a) Possible Reconfiguration Sequences (b) System Modelling through Dynamic Fault Tree's Spare Gates and Components	150
5.2	Graphical Notation of SAN Elements	158

5.3	Atomic Model of Resources (<i>R01_Res1</i>)	162
5.4	Atomic Model of <i>Implementations without Redundancies</i> (<i>I03_FD_Impl1, #3</i>)	164
5.5	Composed Model of <i>Implementations without Redundancies</i> (<i>top_I03_FD_Impl1</i>)	164
5.6	Atomic Model of the <i>Implementations with Redundancies</i> (<i>I01_Impl1, #1</i>)	165
5.7	Composed Model of <i>Implementations with Redundancies</i> (<i>top_I01_Impl1</i>)	165
5.8	Atomic Model of the Reconfiguration Logic (<i>ReconfigurationLogic_SF</i>) .	167
5.9	Atomic Model of the Fail while Active Logic (<i>F01_Impl1_FailActive</i>) . .	168
5.10	Composed Model of the Fail while Active Logic (<i>top_F01_Impl1_FailActive</i>)	168
5.11	Atomic Model of the All Fail Event (<i>F07_SF_AllFail</i>)	169
5.12	Composed Model of the All Fail Event (<i>top_F07_SF_AllFail</i>)	169
5.13	Atomic Model of <i>AllRFailed</i> Event (<i>F03_R_SF_AllRFailed</i>)	170
5.14	Atomic Model of Reconfiguration Implementation's Reconfiguration Logic (<i>ReconfigurationLogic_R_SF</i>)	171
5.15	Composed Model of the All Reconfiguration Implementation Fail Event (<i>top_F03_R_SF_AllRFailed</i>)	171
5.16	Atomic Model of the Reconfiguration Unresolved Event (<i>F04_R_SF_RUnresolved</i>)	172
5.17	Reconfiguration Logic FD_R (<i>ReconfigurationLogic_FD_R_SF</i>)	172
5.18	Composed Model of Reconfiguration Unresolved Event (<i>top_F04_R_SF_RUnresolved</i>)	173
5.19	Atomic Model of the Reconfiguration SF Fail Event (<i>F05_R_SF_Failure</i>)	173
5.20	Composed Model of the Reconfiguration SF Fail Event (<i>top_F05_R_SF_Failure</i>)	173
5.21	Atomic Model of the Control Subfunction Fail Event (<i>I09_ControlSF_NoDependencies</i>)	175
5.22	Composed Model of the Control Subfunction Fail Event (<i>top_I09_ControlSF</i>)	175
5.23	Atomic Model of the Fault Detection Subfunction Failure (<i>F08_FD_SF_Failure</i>)	176
5.24	Composed Model of the Fault Detection Subfunction Failure (<i>top_F08_FD_SF_Failure</i>)	176

5.25	Composed Model of the $\mathcal{F}_{R \text{ Seq.}_1}$ Event (<i>top_F06_Impl1_RF1</i>)	176
5.26	Atomic Model of the $\mathcal{F}_{R \text{ Seq.}_{SF}}$ Event (<i>F06_SF_RF</i>)	177
5.27	Composed Model of the $\mathcal{F}_{R \text{ Seq.}_{SF}}$ Event (<i>top_F06_SF_RF</i>)	177
5.28	Composed Model of the $\mathcal{F}_{FD \text{ Seq.}_1}$ Event	178
5.29	Atomic Model of the $\mathcal{F}_{FD \text{ Seq.}_{SF}}$ Event (<i>F09_FD_SF_FU</i>)	178
5.30	Composed Model of the $\mathcal{F}_{FD \text{ Seq.}}$ Event (<i>top_F09_FD_SF_FU</i>)	178
5.31	Atomic Model of the \mathcal{F}_{SF} Event (<i>F10_SF_Failure</i>)	179
5.32	Composed Model of the \mathcal{F}_{SF} Event (<i>top_F10_SF_Failure</i>)	179
5.33	Atomic Model of the $\mathcal{F}_{ControlSF}$ Event (<i>I09_ControlSF</i>)	181
5.34	Composed Model of the $\mathcal{F}_{ControlSF}$ Event (<i>top_I09_ControlSF</i>)	181
5.35	Atomic Model of the $\mathcal{F}_{OutputSF}$ Event (<i>I10_OutputSF</i>)	182
5.36	Composed Model of the $\mathcal{F}_{OutputSF}$ Event (<i>top_I10_OutputSF</i>)	182
5.37	Normalized Failure Probability of Fire Protection Control Configurations	183
5.38	Fire Protection Control Failure Probability with Ideal Assumptions . . .	187
5.39	Normalized Door Status Control Configurations Failure Probability . . .	189
5.40	Door Status Control Failure Probability with Ideal Assumptions	193
6.1	TCN Configuration Example [IEC07]	199
6.2	<i>TICO</i> Board	200
6.3	<i>CCU/BA</i> Module	200
6.4	<i>MIM</i> Module	201
6.5	Ethernet Switch	202
6.6	Snapshot of the CStools Configuration Software	202
6.7	Tested Real Configuration	203
6.8	Schematic Configuration of the Figure 6.7	203
6.9	Reconfiguration Space of the Tested Scenarios	205
6.10	Reconfiguration Scenarios	206
A.1	Fault Tree Symbols	224
A.2	The BDD of the formula $y = x \wedge y \vee z$	226
A.3	Reliability Block Diagram Structures and Associated Reliability Functions [Alessandro06]	228
A.4	Continuous Time Markov Chain Example	231
A.5	Discrete Time Markov Chain Example	231
A.6	Petri Net Example	234

C.1	Example System: (Static) Fault Tree Model	242
C.2	Example System: Component Fault Tree Model	243
C.3	Example System: HiP-HOPS Model	244
C.4	Dynamic Example System: Dynamic Fault Tree Model	245
C.5	Dynamic Example System: BDMP Model	248
C.6	Dynamic Example System: SEFT Model	249
C.7	Dynamic Example System: SEFT's Underlying DSPN Model	250
D.1	D3H2 Metodology: Design Implementation	254
F.1	Block Diagram of the Repairable PAND Model	268
F.2	Atomic Model of the Component A_BF_B	268
F.3	Atomic Model of the Component A_PAND_B	269
F.4	Repairable PAND gate using Mobius and RAATSS Tools	269

List of Tables

2.1	Fault Classification	21
2.2	Failure/Error Classification	21
2.3	Fault Hypothesis	29
2.4	Failure/Error Model	29
2.5	Limitations of Event-Based Approaches [Aizpurua13b]	33
2.6	Addressed Characteristics by the Analysed Approaches	47
2.7	Approaches and Addressed Design Properties	61
2.8	Design Decisions and Influenced Attributes	63
3.1	Possible Compatible Physical Locations	73
3.2	Comparison of Redundancies with respect to the Nominal Configuration	75
3.3	Reconfiguration Table Example	77
3.4	Functional Model for Air Conditioning Control in Train.Car ₁	82
3.5	Preliminary HW/SW Architecture for Air Conditioning Control in Train.Car ₁ .Zone _A	84
3.6	Extended HW/SW Architecture for the Air Conditioning Control Main Function in Train.Car ₁ .Zone _A	85
3.7	Reconfiguration Table for the Air Conditioning Control Main Function in Train.Car ₁ .Zone _A	86
3.8	Functional Model for the Functions in Train.Car ₁ .Zone _A	87
3.9	Preliminary HW/SW Architecture for the Fire Protection Control in Train.Car ₁ .Zone _A	88
3.10	Extended HW/SW Architecture for the Fire Protection Control in Train.Car ₁ .Zone _A	89
3.11	Reconfiguration Table of the Fire Protection Main Function in the Train.Car ₁ .Zone _A	90
3.12	Functional Model for the Functions in the Train.Car ₁ .Zone _A .Door	91

3.13	Preliminary HW/SW Architecture for the Door Status Control in the Train.Car ₁ .Zone _A .Door	93
3.14	Extended HW/SW Architecture for the Door Status Control in the Train.Car ₁ .Zone _A .Door	95
3.15	Reconfiguration Table of the Door Status Control Main Function in the Train.Car ₁ .Zone _A .Door	96
4.1	Notation of Failure and Working Events	103
4.2	Approach and Characteristics	108
4.3	Component Dynamic Fault Tree Gates	111
4.4	Fire Protection Control Configurations with Alternative Redundancy Strategies	127
4.5	Failure Criticality Index Values of the Fire Protection Control (10 ⁶ iterations)	130
4.6	Unreliability and FCI values for Fire Protection Control Configurations under Different Assumptions (10 ⁶ iterations)	132
4.7	Door Status Control Configurations with Alternative Redundancy Strategies	138
4.8	Door Status Control Failure Probability for Reconfiguration Distribution Strategies (T=10 years)	141
4.9	$\mathcal{FCI}_{\mathcal{F}_{FD_SF}}$ and $\mathcal{FCI}_{\mathcal{F}_{R_SF}}$ using Different Redundancy Strategies (10 ⁶ iterations)	143
4.10	Failure Probabilities and FCI Values for Configurations under Different Assumptions (10 ⁶ iterations)	144
5.1	Notation of Failure and Working Events II	153
5.2	Repairable HW/SW Architecture Example	162
5.3	Activities in <i>I03_FD_Impl1</i>	164
5.4	Activities in <i>I01_Impl1</i>	165
5.5	Activities in <i>ReconfigurationLogic_SF</i>	167
5.6	Activities in <i>F01_Impl1_FailActive</i>	168
5.7	Fault Detection and Reconfiguration Failure Events and Assigned Names	168
5.8	Activities in <i>F07_SF_AllFail</i>	169
5.9	Activities in <i>F03_R_SF_AllRFailed</i>	170
5.10	Activities in <i>ReconfigurationLogic_R_SF</i>	171

5.11	Activities in <i>F04_R_SF_RUnresolved</i>	172
5.12	<i>ReconfigurationLogic_FD_R_SF</i> Activity Characterization	172
5.13	Activities in <i>F05_R_SF_Failure</i>	173
5.14	Activities in <i>I09_ControlSF_NoDependencies</i>	175
5.15	Activities in <i>F08_FD_SF_Failure</i>	176
5.16	Activities in <i>F06_SF_RF</i>	177
5.17	Activities in <i>F09_FD_SF_FU</i>	178
5.18	Activities in <i>F10_SF_Failure</i>	179
5.19	Activities in <i>I09_Control_SF_Failure</i>	181
5.20	Activities in <i>I10_OutputSF_Failure</i>	182
5.21	Normalized Cost of Alternative Fire Protection Control Configurations .	184
5.22	Fire Protection Control (FPC) Unavailability for Reconfiguration Distri- bution Strategies (T=10 years)	185
5.23	Failure Probability of the Fire Detection and its Underlying Events (T=10 years)	186
5.24	Normalized Cost of Alternative Door Status Control Configurations . . .	190
5.25	Door Status Control (DSC) Unavailability for Reconfiguration Distribu- tion Strategies (T=10 years)	191
5.26	Failure Probability of the Underlying Events of the Door Status Control Main Function (T=10 years)	192
B.1	Summary of Limitations Overcome by Approaches	237
B.2	Tool-Support of the Dynamic Approaches	239
B.3	Tool-Support of the CFP Approaches	240
B.4	Tool-Support of the Transformational Approaches	240
E.1	Failure Rates & Cost Values of HW, SW and Communication Resources	263
F.1	Activities in the SAN model <i>A_BF_B</i>	268
F.2	Activities in the SAN model <i>A_PAND_B</i>	269

List of Algorithms

1	Criticality Analysis	118
2	Heterogeneous Redundancy Identification (Natural Compatibility)	258
3	Heterogeneous Redundancy Suggestion (Forced Compatibility)	259
4	Reconfiguration Table Extraction Algorithm	261

Glossary

Adaptation The ability of a system to adapt itself to its environment. 3, 52, 54, 55, 57, 63

Architecture Allocation of software functions onto available hardware resources satisfying functional and dependability requirements. 3, 6–9, 24, 27, 32, 33, 40, 43–46, 49–51, 53, 54, 56, 62, 63, 65, 66, 68, 69, 76, 77, 79, 80, 82, 83, 88, 92, 94, 96, 99–102, 104, 123, 127–130, 138, 140–143, 145–147, 149, 151, 152, 154, 157, 161, 182, 183, 188, 189, 195–198, 200, 203, 209, 212, 214, 216, 217

Configuration a possible realization of the main function comprised of the necessary subfunctions and their underlying implementations (and resources) to perform the main function. 71, 75, 76, 92, 127–130, 132, 133, 138, 139, 141–145, 155, 156, 172, 177, 182–196, 202, 203, 213–215

Dependability Ability to avoid failures that are more severe and more frequent than is acceptable. 2, 3, 5–8, 57–60, 62–66, 68, 99–102, 104, 107, 119, 123, 133, 140, 142, 145, 147, 149, 152, 154, 157, 161, 194–196, 211–213, 215–217

Design A specification of a system intended to accomplish goals in a particular environment, using a set of components, satisfying a set of requirements, subject to constraints. 1–3, 6–8, 11, 18, 24, 27, 28, 30–34, 40, 41, 44–46, 48–57, 59, 60, 62–68, 72, 75, 77–80, 83, 93, 94, 96, 99–102, 104, 107, 115, 122, 123, 133, 138, 139, 147, 151, 152, 154, 156, 157, 194–198, 204, 207, 211–213, 216, 217

Fault-Tolerance Mechanisms to avoid system failures in the presence of faults. 1, 24, 26–28, 30–32, 49, 66, 76, 209

Function What the system is intended to do. 2–8, 12, 14, 16, 17, 21, 22, 25, 31, 32, 50, 51, 58, 60, 63, 64, 66–72, 75–83, 86, 87, 90, 92, 96, 100, 102, 111, 120, 123, 133, 137, 147, 161, 174, 181, 186, 197–199, 201, 204, 206, 209, 212–215, 217, 218

Health Management Reconfiguration and fault detection implementations which make possible to manage the system’s behaviour in the presence of failures. 3, 6, 53, 65, 66, 68, 76–79, 83, 88, 92, 100, 101, 130, 140, 145, 184–186, 188, 190–193, 196, 212, 215, 216

Heterogeneous Redundancy Redundancies which reuse existing hardware resources and provide compatible functionality e.g., analytical redundancy. 3, 31, 62, 76, 88, 100, 128, 130, 132, 138, 139, 143, 146, 147, 184, 187, 190, 195, 212, 214, 215

Homogeneous Redundancy Redundancies which replicate the nominal functionality making use of additional hardware components. 82, 128, 138, 139, 143, 183, 189, 215

Reconfiguration The process through which a system halts operation under its current source configuration and begins to operate under a different target configuration. 3, 6–8, 26, 31, 53–56, 59, 60, 62–66, 68, 69, 76–80, 83, 84, 86, 89, 90, 92–94, 96, 97, 100–102, 104–107, 122–125, 128–131, 133–135, 138, 140–147, 150, 153–157, 165–173, 175–177, 182–188, 190–198, 203–208, 212–215, 217

Resource A hardware, software or communication device which is able to perform a function in conjunction with other devices or by itself. 1–7, 12, 14, 25, 26, 29–31, 33, 49–52, 54, 56, 57, 63, 66–72, 75, 77, 79, 80, 83, 96, 101, 102, 104, 115, 122, 123, 126, 128, 129, 132, 134, 136, 140, 144, 146, 147, 149–153, 161–167, 169, 173, 174, 180–182, 184, 188, 190, 193–195, 197, 198, 212–215, 217, 218

System set of mutually related elements or parts assembled together in some specified order to perform an intended function. 1–3, 5–9, 11, 13, 18, 28, 29, 31, 33–36, 39, 40, 42–46, 48–60, 62–73, 75–77, 79, 83, 96, 97, 99, 104, 109, 114, 116, 129–133, 139–147, 149–158, 160, 161, 167, 175, 182, 184, 185, 188, 190, 191, 194–200, 209, 211–218

Bibliography

- [Adachi11] M. Adachi, Y. Papadopoulos, S. Sharvia, D. Parker, and T. Tohdo, “An approach to optimization of fault tolerant architectures using HiP-HOPS”, *Softw. Pract. Exp.*, 2011.
- [Adler08] R. Adler, D. J. Domis, M. Forster, and M. Trapp, “Probabilistic analysis of safety-critical adaptive systems with temporal dependences”, in *Proc. of RAMS’08*, pp. 149–154, IEEE, 2008.
- [Adler10a] R. Adler, D. Domis, K. Höfig, S. Kemmann, T. Kuhn, J. Schwinn, and M. Trapp, “Integration of component fault trees into the UML”, in *MoDELS’10*, pp. 312–327, 2010.
- [Adler10b] R. Adler, I. Schaefer, M. Trapp, and A. Poetzsch-Heffter, “Component-based modeling and verification of dynamic adaptation in safety-critical embedded systems”, *ACM Trans. Embed. Comput. Syst.*, vol. 10, no. 2, 20:1–20:39, Dec. 2010.
- [Adler10c] R. Adler, D. Schneider, and M. Trapp, “Engineering dynamic adaptation for achieving cost-efficient resilience in software-intensive embedded systems”, in *Proc. of Engineering of Complex Computer Systems*, pp. 21–30, IEEE, 2010.
- [Agrawal88] P. Agrawal, “Fault Tolerance in Multiprocessor Systems Without Dedicated Redundancy”, *IEEE Trans. Comput.*, vol. 37, no. 3, 358–362, Mar. 1988.
- [Aizpurua12a] J. I. Aizpurua and E. Muxika, “Dependable Design: Trade-Off Between the Homogeneity and Heterogeneity of Functions

- and Resources”, in *Proceedings of DEPEND 2012*, pp. 13–17, 2012.
- [Aizpurua12b] J. I. Aizpurua and E. Muxika, “Design of Dependable Systems: An Overview of Analysis and Verification Approaches”, in *Proceedings of DEPEND 2012*, pp. 4–12, 2012.
- [Aizpurua13a] J. I. Aizpurua and E. Muxika, “Functionality and Dependability Assurance in Massively Networked Scenarios”, in *Safety, Reliability and Risk Analysis: Beyond the Horizon*, pp. 1763 — 1771, CRC Press, 2013.
- [Aizpurua13b] J. I. Aizpurua and E. Muxika, “Model Based Design of Dependable Systems: Limitations and Evolution of Analysis and Verification Approaches”, *International Journal on Advances in Security*, vol. 6, 12–31, 2013.
- [Aizpurua14] J. I. Aizpurua, E. Muxika, G. Manno, and F. Chiacchio, “Heterogeneous Redundancy Analysis based on Component Dynamic Fault Trees”, in *Proceedings of PSAM 12*, 2014.
- [Alessandro06] B. Alessandro, *Reliability Engineering: Theory and Practice*, Springer, 2006.
- [Ana10] “Oscillatory failure case detection in the A380 electrical flight control system by analytical redundancy”, *Control Engineering Practice*, vol. 18, no. 9, 1110 – 1119, 2010.
- [Arnold99] A. Arnold, G. Point, A. Griffault, and A. Rauzy, “The AltaRica formalism for describing concurrent systems”, *Fundamenta Informaticae*, vol. 40, no. 2-3, 109–124, 1999.
- [Arnold13] F. Arnold, A. F. E. Belinfante, F. I. Van der Berg, D. Guck, and M. I. A. Stoelinga, “DFTCalc: a tool for efficient fault tree analysis (extended version)”, Technical Report TR-CTIT-13-13, Centre for Telematics and Information Technology, University of Twente, Enschede, June 2013.

- [ATESST10] ATESSST, “ATESST2 Homepage”, 2010. Online. Accessed on 06/10/2014. Available at: <http://www.east-adl.fr>.
- [Avizienis85] A. Avizienis, “The N-Version Approach to Fault-Tolerant Software”, *Software Engineering, IEEE Transactions on*, vol. SE-11, no. 12, 1491–1501, Dec 1985.
- [Avizienis04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic Concepts and Taxonomy of Dependable and Secure Computing”, *IEEE Trans. Dependable Secur. Comput.*, vol. 1, 11–33, 2004.
- [Batteux13] M. Batteux, T. Prosvirnova, A. Rauzy, and L. Kloul, “The AltaRica 3.0 project for model-based safety assessment”, in *Industrial Informatics (INDIN), 2013 11th IEEE International Conference on*, pp. 741–746, July 2013.
- [Bause02] F. Bause and P. S. Kritzinger, *Stochastic Petri nets - an introduction to the theory.*, Advanced studies of computer science, Vieweg, 2002.
- [Bernardi12] S. Bernardi, J. Merseguer, and D. Petriu, “Dependability modeling and analysis of software systems specified with UML”, *ACM Computing Survey*, vol. 45, no. 1, 2, 2012.
- [Bieber02] P. Bieber, C. Castel, and C. Seguin, “Combination of Fault Tree Analysis and Model Checking for Safety Assessment of Complex System”, in *Proc. of EDCC’02*, vol. 2485, pp. 624–628, Springer, 2002.
- [Bieber09] P. Bieber, E. Noulard, C. Pagetti, T. Planche, and F. Vialard, “Preliminary design of future reconfigurable IMA platforms”, *SIGBED Rev.*, vol. 6, no. 3, 2009.
- [Bieber10] P. Bieber, J. Brunel, E. Noulard, C. Pagetti, T. Planche, and F. Vialard, “Preliminary Design of Future Reconfigurable IMA Platforms - Safety Assessment”, in *27th International Congress of the Aeronautical Sciences*, 2010.

- [Biehl10] M. Biehl, C. DeJiu, and M. Törngren, “Integrating safety analysis into the model-based development toolchain of automotive embedded systems”, in *Proc. of LCTES '10*, pp. 125–132, ACM, 2010.
- [Blanke11] M. Blanke, S. Hansen, and M. R. Blas, “Diagnosis for Control and Decision Support in Complex Systems”, in *Proceedings Volume from the Special International Conference on Complex Systems*, pp. 89–101, 2011.
- [Bobbio04] A. Bobbio and D. Raiteri, “Parametric fault trees with dynamic gates and repair boxes”, in *Reliability and Maintainability, 2004 Annual Symposium - RAMS*, pp. 459–465, Jan 2004.
- [Bondavalli90] A. Bondavalli and L. Simoncini, “Failure classification with respect to detection”, in *Proceedings of the 2nd IEEE Workshop on Future Trends of Distributed Proceedings*, pp. 47–53, Publ by IEEE, Cairo, Egypt, 1990.
- [Bouissou07] M. Bouissou, “A generalization of Dynamic Fault Trees through Boolean logic Driven Markov Processes (BDMP)”, in *Proc. of ESREL'07*, vol. 2, pp. 1051–1058, 2007.
- [Bryant86] R. Bryant, “Graph Based Algorithms for Boolean Function Manipulation”, *IEEE Transactions on Computers*, vol. C-35, 677–691, 1986.
- [Burlando92] P. Burlando, L. Gianetto, and M. Mainini, “Functional Diversity”, vol. 1 of *Research Reports ESPRIT*, pp. 49–113, Springer Berlin Heidelberg, 1992.
- [Cauffriez04] L. Cauffriez, J. Ciccotelli, B. Conrard, and M. Bayart, “Design of intelligent distributed control systems: a dependability point of view”, *Reliability Engineering & System Safety*, vol. 84, no. 1, 19–32, 2004.
- [Cauffriez13] L. Cauffriez, D. Renaux, T. Bonte, and E. Cocquebert, “Sys-

- temic Modeling of Integrated Systems for Decision Making Early on in the Design Process.”, *Cybernetics and Systems*, vol. 44, 1–22, 2013.
- [CHESS12] CHESS, “CHESS Project”, 2012. Online. Accessed on 06/10/2014. Available at: <http://www.chess-project.org/>.
- [Chiacchio11] F. Chiacchio, L. Compagno, D. D’Urso, G. Manno, and N. Trapani, “Dynamic fault trees resolution: A conscious trade-off between analytical and simulative approaches”, *Reliability Engineering and System Safety*, vol. 96, no. 11, 1515–1526, 2011.
- [Chiola93a] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad, “Stochastic well-formed colored nets and symmetric modeling applications”, *Computers, IEEE Transactions on*, vol. 42, no. 11, 1343–1360, Nov 1993.
- [Chiola93b] G. Chiola and A. Ferscha, “Distributed simulation of timed Petri nets: Exploiting the net structure to obtain efficiency”, in M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, vol. 691 of *Lecture Notes in Computer Science*, pp. 146–165, Springer Berlin Heidelberg, 1993.
- [Choi94] H. Choi, V. G. Kulkarni, and K. S. Trivedi, “Markov Regenerative Stochastic Petri Nets.”, *Performance Evaluation*, vol. 20, no. 1-3, 337–357, 1994.
- [Clarhaut09] J. Clarhaut, S. Hayat, B. Conrard, and V. Cocquempot, “Optimal design of dependable control system architectures using temporal sequences of failures”, *IEEE Transactions On Reliability*, vol. 58, no. 3, 511–522, 2009.
- [Clements01] P. C. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, SEI Series in Software Engineering, Addison-Wesley, 2001.

- [CMU12] CMU, “OSATE”, 2012. Online. Accessed on 06/10/2014. Available at: https://wiki.sei.cmu.edu/aadl/index.php/AADL_tools.
- [Codetta-Raiteri05] D. Codetta-Raiteri, “The Conversion of Dynamic Fault Trees to Stochastic Petri Nets, as a case of Graph Transformation”, *Electronic Notes in Theoretical Computer Science*, vol. 127, no. 2, 45 – 60, 2005.
- [Cortellessa06] V. Cortellessa, F. Marinelli, and P. Potena, “Automated selection of software components based on cost/reliability trade-off”, in *Software Architecture*, pp. 66–81, Springer, 2006.
- [Courtney04] T. Courtney, D. Daly, S. Derisavi, S. Gaonkar, M. Griffith, V. Lam, and W. Sanders, “The Mobius modeling environment: recent developments”, in *Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings. First International Conference on the*, pp. 328–329, Sept 2004.
- [Cressent11] R. Cressent, V. Idasiak, F. Kratz, and P. David, “Mastering safety and reliability in a Model Based process”, in *Proc. of RAMS’11*, 2011.
- [Crnkovic03] I. Crnkovic, “Component-based software engineering - new challenges in software development”, in *Information Technology Interfaces, 2003. ITI 2003. Proceedings of the 25th International Conference on*, pp. 9–18, 2003.
- [Distefano07] S. Distefano and A. Puliafito, “Dynamic Reliability Block Diagrams VS Dynamic Fault Trees”, *In Proc. of RAMS’07*, vol. 8, 71–76, 2007.
- [Distefano09] S. Distefano and A. Puliafito, “Dependability Evaluation with Dynamic Reliability Block Diagrams and Dynamic Fault Trees.”, *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 1, 4–17, 2009.
- [Domis09a] D. Domis, M. Forster, S. Kemmann, and M. Trapp, “Safety

- Concept Trees”, in *Reliability and Maintainability Symposium, 2009. RAMS 2009. Annual*, pp. 212–217, Jan 2009.
- [Domis09b] D. Domis and M. Trapp, “Component-Based Abstraction in Fault Tree Analysis”, in *Computer Safety, Reliability, and Security*, vol. 5775 of *LNI*, pp. 297–310, Springer, 2009.
- [Doyle95] S. A. Doyle and J. B. Dugan, “Dependability Assessment using Binary Decision Diagrams (BDDs)”, in *FTCS*, pp. 249–258, 1995.
- [Dugan92] J. Dugan, S. Bavuso, and M. Boyd, “Dynamic fault-tree models for fault-tolerant computer systems”, *IEEE Trans. on Reliability*, vol. 41, no. 3, 363–377, 1992.
- [Eclipse12] Eclipse, “Eclipse Papyrus”, 2012. Online. Accessed on 06/10/2014. Available at: <http://www.eclipse.org/papyrus/>.
- [EDF14] EDF, “KB3 Workbench”, 2014. Online. Accessed on 06/10/2014. Available at: <http://sourceforge.net/projects/visualfigaro/files/>.
- [Edifor12] E. Edifor, M. Walker, and N. A. Gordon, “Quantification of Priority-OR Gates in Temporal Fault Trees.”, in F. Ortmeier and P. Daniel, editors, *SAFECOMP*, vol. 7612 of *Lecture Notes in Computer Science*, pp. 99–110, Springer, 2012.
- [Edifor13] E. Edifor, M. Walker, and N. A. Gordon, “Quantification of Simultaneous-AND Gates in Temporal Fault Trees”, in *New Results in Dependability and Computer Systems*, vol. 224, pp. 141–151, Springer, 2013.
- [Elegbede03] A. Elegbede, C. Chu, K. Adjallah, and F. Yalaoui, “Reliability allocation through cost minimization”, *Reliability, IEEE Transactions on*, vol. 52, no. 1, 106–111, March 2003.
- [Engel10] C. Engel, A. Roth, P. H. Schmitt, R. Coutinho, and

- T. Schoofs, “Enhanced dispatchability of aircrafts using multi-static configurations”, in *Proc. of ERTS’10*, 2010.
- [Feiler07] P. Feiler and A. Rugina, “Dependability Modeling with the Architecture Analysis & Design Language (AADL)”, Technical Note CMU/SEI-2007-TN-043, CMU Software Engineering Institute, 2007, 2007.
- [Fenelon93] P. Fenelon and J. A. McDermid, “An integrated tool set for software safety analysis”, *J. Syst. Softw.*, vol. 21, 279–290, 1993.
- [Flammini09] F. Flammini, A. Gaglione, N. Mazzocca, V. Moscato, and C. Pragliola, “On-line integration and reasoning of multi-sensor data to enhance infrastructure surveillance”, *Journal of Information Assurance and Security*, vol. 4, 183–191, 2009.
- [Flammini10] F. Flammini, A. Gaglione, F. Ottello, A. Pappalardo, C. Pragliola, and A. Tedesco, “Towards Wireless Sensor Networks for railway infrastructure monitoring”, in *Electrical Systems for Aircraft, Railway and Ship Propulsion (ESARS), 2010*, pp. 1–6, Oct 2010.
- [Flammini11] F. Flammini, N. Mazzocca, A. Pappalardo, C. Pragliola, and V. Vittorini, “Augmenting Surveillance System Capabilities by Exploiting Event Correlation and Distributed Attack Detection”, in *Proceedings of the IFIP WG 8.4/8.9 International Cross Domain Conference on Availability, Reliability and Security for Business, Enterprise and Health Information Systems, ARES’11*, pp. 191–204, Springer-Verlag, Berlin, Heidelberg, 2011.
- [Forster09] M. Forster and M. Trapp, “Fault Tree Analysis of Software-Controlled Component Systems Based on Second-Order Probabilities”, in *International Symposium on Software Reliability Engineering - ISSRE’09*, pp. 146–154, IEEE Computer Society, 2009.

- [Forster10] M. Forster and D. Schneider, “Flexible, Any-Time Fault Tree Analysis with Component Logic Models”, in *ISSRE*, pp. 51–60, IEEE Computer Society, 2010.
- [Fricks03] R. M. Fricks and K. S. Trivedi, “Importance analysis with Markov chains”, in *Reliability and Maintainability Annual Symposium*, pp. 89–95, 2003.
- [Fuentes04] L. Fuentes and A. Vallecillo, “An Introduction to UML Profiles”, *Journal of UML and Model Engineering*, vol. 5, no. 2, 5–13, 2004.
- [Galdun08] J. Galdun, J. Ligus, J.-M. Thiriet, and J. Sarnovsky, “Reliability increasing through networked cascade control structure - consideration of quasi-redundant subsystems”, in *IFAC Proc. Volumes*, vol. 17, pp. 6839–6844, 2008.
- [Gallina12] B. Gallina, M. A. Javed, F. U. Muram, and S. Punnekkat, “Model-driven Dependability Analysis Method for Component-based Architectures”, in *Euromicro-SEAA Conference*, IEEE Computer Society, 2012.
- [Galloway02] A. Galloway, J. McDermid, J. Murdoch, and D. Pumfrey, “Automation of System Safety Analysis: Possibilities and Pitfalls”, in *Proc. of ISSC’02*, 2002.
- [Goševa-Popstojanova01] K. Goševa-Popstojanova and K. S. Trivedi, “Architecture-based approach to reliability assessment of software systems”, *Performance Evaluation*, vol. 45, no. 2–3, 179 – 204, 2001.
- [Gokhale07] S. S. Gokhale et al., “Architecture-based software reliability analysis: Overview and limitations”, *IEEE Transactions on dependable and secure computing*, vol. 4, no. 1, 32–40, 2007.
- [Gouberman14] A. Gouberman, C. Grand, M. Riedl, and M. Siegle, “An IDE for the LARES Toolset”, in K. Fischbach and U. R. Krieger, editors, *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance - 17th In-*

ternational GI/ITG Conference, vol. 8376 of *Lecture Notes in Computer Science*, pp. 240–254, Springer, 2014.

- [Harel87] D. Harel, “Statecharts: A Visual Formalism For Complex Systems”, 1987.
- [Haverkort01] B. R. Haverkort, “Markovian models for performance and dependability evaluation”, in *Lectures on Formal Methods and Performance Analysis*, pp. 38–83, Springer, 2001.
- [Henk C.03] T. Henk C., *A First Course in Stochastic Models*, Wiley, 2003.
- [Hilber05] P. Hilber and L. Bertling, “A method for extracting reliability importance indices from reliability simulations of electrical networks”, *Proc. 15th Power Systems Computation Conference (PSCC)*, 2005.
- [Hoftberger13] O. Hoftberger and R. Obermaisser, “Ontology-based Runtime Reconfiguration of Distributed Embedded Real-Time Systems”, in *The 16th IEEE Computer Society symposium on object/component/service-oriented realtime distributed computing (ISORC 2013)*, IEEE, 2013.
- [Hull14] U. Hull, “HiP-HOPS Tool”, 2014. Online. Accessed on 06/10/2014. Available at: <http://hip-hops.eu/>.
- [IAEA88] IAEA, “Component Reliability Data For Use In Probabilistic Safety Assessment, IAEA-TECDOC-478”, Tech. rep., 1988.
- [IEC07] IEC, “Train Communication Network, IEC 61375”, Tech. rep., 2007.
- [Illinois14] U. Illinois, “MOBIUS Tool - Model based environment for validation of system reliability, availability, security, and performance”, 2014. Online. Accessed on 06/10/2014. Available at: <https://www.mobius.illinois.edu/>.
- [Ilmenau07] T. Ilmenau, “TimeNET Tool”, 2007. Online. Accessed on

06/10/2014. Available at: <http://www.tu-ilmenau.de/sse/timenet/>.

- [Isermann05] R. Isermann, “Model-based fault-detection and diagnosis—status and applications”, *Annual Reviews in control*, vol. 29, no. 1, 71–85, 2005.
- [Izosimov05] V. Izosimov, P. Pop, P. Eles, and Z. Peng, “Design optimization of time- and cost-constrained fault-tolerant distributed embedded systems”, in *Design, Automation and Test in Europe, 2005. Proceedings*, pp. 864–869 Vol. 2, March 2005.
- [Jean-Pascal13] S. Jean-Pascal, A. Rasmus, and K. Sören, “Combining Safety Engineering and Product Line Engineering”, in *Software Engineering (Workshops)*, pp. 545–554, 2013.
- [Johnson84] B. W. Johnson, “Fault-Tolerant Microprocessor-Based Systems”, *IEEE Micro*, vol. 4, no. 6, 6–21, Dec. 1984.
- [Joshi07] A. Joshi, S. Vestal, and P. Binns, “Automatic Generation of Static Fault Trees from AADL models”, in *DNS Workshop on Architecting Dependable Systems*, Springer, 2007.
- [jvc] “JVC Professional”, 2014. Online. Accessed on 06/10/2014. Available at: <http://pro.jvc.com/>.
- [Kaaniche02] M. Kaaniche, J.-C. Laprie, and J.-P. Blanquart, “A framework for dependability engineering of critical computing systems”, *Safety Science*, vol. 40, no. 9, 731–752, 2002.
- [Kaiser03] B. Kaiser, P. Liggesmeyer, and O. Mäckel, “A new component concept for fault trees”, in *Proc. of SCS'03*, pp. 37–46, 2003.
- [Kaiser07] B. Kaiser, C. Gramlich, and M. Forster, “State-Event Fault Trees - A Safety Analysis Model for Software-Controlled Systems”, *Reliability Eng. System Safety*, vol. 92, no. 11, 1521–1537, 2007.

- [Kanoun01] K. Kanoun, “Real-world design diversity: a case study on cost”, *Software, IEEE*, vol. 18, no. 4, 29–33, Jul 2001.
- [Kartson94] D. Kartson, G. Balbo, S. Donatelli, G. Franceschinis, and G. Conte, *Modelling with Generalized Stochastic Petri Nets*, John Wiley & Sons, Inc., New York, NY, USA, 1st edn., 1994.
- [Knight86] J. C. Knight and N. G. Leveson, “An Experimental Evaluation Of The Assumption Of Independence In Multi-Version Programming”, *IEEE Transactions on Software Engineering*, vol. 12, 96–109, 1986.
- [Krysander08] M. Krysander, J. Aslund, and M. Nyberg, “An Efficient Algorithm for Finding Minimal Overconstrained Subsystems for Model-Based Diagnosis”, *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 38, no. 1, 197–206, Jan 2008.
- [Labri14] Labri, “AltaRica Tool”, 2014. Online. Accessed on 06/10/2014. Available at: <http://altarica.labri.fr/wp/>.
- [Laprie92] J. C. Laprie, A. Avizienis, and H. Kopetz, editors, *Dependability: Basic Concepts and Terminology*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.
- [Laprie95] J.-C. Laprie, J. Arlat, C. Beounes, and K. Kanoun, “Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures”, in B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood, editors, *Predictably Dependable Computing Systems*, ESPRIT Basic Research Series, pp. 103–122, Springer Berlin Heidelberg, 1995.
- [Leveson95] N. G. Leveson, *Safeware - system safety and computers: a guide to preventing accidents and losses caused by technology*, Addison-Wesley, 1995.
- [Lindemann98] C. Lindemann, *Performance modelling with deterministic and*

stochastic Petri nets, Wiley-Interscience series in systems and optimization, Wiley, 1998.

- [Lisagor10] O. Lisagor, *Failure Logic Modelling: A Pragmatic Approach*, Ph.D. thesis, Department of Computer Science, The University of York, 2010.
- [Littlewood96] B. Littlewood, “The impact of diversity upon common mode failures”, *Reliability Engineering and System Safety*, vol. 51, 101–113, 1996.
- [Littlewood00a] B. Littlewood and L. Strigini, “A discussion of practices for enhancing diversity in software designs”, Tech. rep., Centre for Software Reliability, City University, 2000.
- [Littlewood00b] B. Littlewood and L. Strigini, “Software Reliability and Dependability: A Roadmap”, in *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pp. 175–188, ACM, New York, NY, USA, 2000.
- [Littlewood01a] B. Littlewood, P. Popov, and L. Strigini, “Design Diversity: an Update from Research on Reliability Modelling”, in F. Redmill and T. Anderson, editors, *Aspects of Safety Management*, pp. 139–154, Springer London, 2001.
- [Littlewood01b] B. Littlewood, P. Popov, and L. Strigini, “Modeling Software Design Diversity: A Review”, *ACM Comput. Surv.*, vol. 33, no. 2, 177–208, Jun. 2001.
- [Lopatkin11] I. Lopatkin, A. Iliasov, and A. Romanovsky, “Rigorous Development of Dependable Systems Using Fault Tolerance Views”, in *Proceedings of the 2011 IEEE 22nd International Symposium on Software Reliability Engineering*, ISSRE '11, pp. 180–189, IEEE Computer Society, 2011.
- [Lyu07] M. R. Lyu, “Software Reliability Engineering: A Roadmap”, in *Future of Software Engineering, 2007. FOSE '07*, pp. 153–170, 2007.

- [Mahmud12] N. Mahmud, M. Walker, and Y. Papadopoulos, “Compositional Synthesis of Temporal Fault Trees from State Machines”, *SIGMETRICS Perform. Eval. Rev.*, vol. 39, no. 4, 79–88, Apr. 2012.
- [Manno12a] G. Manno, *Reliability Modelling of Complex Systems: and adaptive transition system to match accuracy and efficiency*, Ph.D. thesis, University of Catania, 2012.
- [Manno12b] G. Manno, F. Chiacchio, L. Compagno, D. D’Urso, and N. Trapani, “MatCarloRe: An integrated FT and Monte Carlo Simulink tool for the reliability assessment of dynamic fault tree”, *Expert Systems with Applications*, vol. 39, no. 12, 10334–10342, 2012.
- [Manno14a] G. Manno and F. Chiacchio, “MatCarloRE Tool”, 2014. Online. Accessed on 06/10/2014. Available at: <http://www.dmi.unict.it/~chiacchio/?m=5&project=matcarlo>.
- [Manno14b] G. Manno and F. Chiacchio, “RAATSS Tool”, 2014. Online. Accessed on 06/10/2014. Available at: <http://www.dmi.unict.it/~chiacchio/?m=5&project=raatss>.
- [Manno14c] G. Manno, F. Chiacchio, L. Compagno, D. D’Urso, and N. Trapani, “Conception of Repairable Dynamic Fault Trees and resolution by the use of RAATSS, a Matlab® toolbox based on the ATS formalism”, *Reliability Engineering & System Safety*, vol. 121, no. 0, 250 – 262, 2014.
- [MathWorks14] MathWorks, “Matlab/Simulink”, 2014. Online. Accessed on 06/10/2014. Available at: <http://www.mathworks.com>.
- [Medvidovic00] N. Medvidovic and R. N. Taylor, “A Classification and Comparison Framework for Software Architecture Description Languages”, *IEEE Trans. Softw. Eng.*, vol. 26, no. 1, 70–93, Jan. 2000.
- [Meedeniya11] I. Meedeniya, I. Moser, A. Aleti, and L. Grunske,

- “Architecture-based Reliability Evaluation Under Uncertainty”, in *Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS*, QoSA-ISARCS ’11, pp. 85–94, ACM, 2011.
- [Merle10] G. Merle, J.-M. Roussel, J.-J. Lesage, and A. Bobbio, “Probabilistic Algebraic Analysis of Fault Trees With Priority Dynamic Gates and Repeated Events.”, *IEEE Trans. on Reliability*, vol. 59, no. 1, 250–261, 2010.
- [Merle14] G. Merle, J.-M. Roussel, and J.-J. Lesage, “Quantitative Analysis of Dynamic Fault Trees Based on the Structure Function”, *Quality and Reliability Eng. Int.*, vol. 30, no. 1, 143–156, 2014.
- [Montani08] S. Montani, L. Portinale, A. Bobbio, and D. Codetta-Raiteri, “Radyban: A tool for reliability analysis of dynamic fault trees through conversion into dynamic Bayesian networks”, *Reliability Engineering & System Safety*, vol. 93, no. 7, 922 – 932, 2008.
- [Montecchi11] L. Montecchi, P. Lollini, and A. Bondavalli, “An Intermediate Dependability Model for state-based dependability analysis”, Tech. rep., University of Florence, Dip. Sistemi Informatica, RCL group, 2011.
- [Moore01] J. Moore, *The Avionics Handbook*, chap. Advanced Distributed Architectures, CRC Press, 2001.
- [Nelson90] V. P. Nelson, “Fault-Tolerant Computing: Fundamental Concepts”, *Computer*, vol. 23, 19–25, July 1990.
- [Niu11] R. Niu, T. Tang, O. Lisagor, and J. McDermid, “Automatic safety analysis of networked control system based on failure propagation model”, in *Proc. of ICVES’11*, pp. 53–58, 2011.
- [Nord03] R. L. Nord, M. R. Barbacci, P. Clements, R. Kaz-

- man, and M. Klein, “Integrating the Architecture Trade-off Analysis Method (ATAM) with the cost benefit analysis method (CBAM)”, Tech. rep., Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, Technical Note CMU/SEI-2003-TN-038, 2003.
- [Oberkampff04] W. Oberkampff, J. C. Helton, C. A. Joslyn, S. F. Wojtkiewicz, and S. Ferson, “Challenge problems: uncertainty in system response given uncertain parameters.”, *Reliability Engineering & System Safety*, vol. 85, no. 1-3, 11–19, 2004.
- [Office02] N. Office, M. Assurance, and N. Headquarters, “Fault Tree Handbook with Aerospace Applications”, *Director*, p. 218, 2002.
- [OMG03] OMG, “MDA Guide Version 1.0.1”, 2003. Online. Accessed on 06/10/2014. Available at: <http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf>.
- [OMG14a] OMG, “Systems Modelling Language”, 2014. Online. Accessed on 06/10/2014. Available at: <http://www.omgsysml.org/>.
- [OMG14b] OMG, “The Unified Modeling Language”, 2014. Online. Accessed on 06/10/2014. Available at: <http://www.uml.org/>.
- [Ou00] Y. Ou and J. B. Dugan, “Sensitivity Analysis of Modular Dynamic Fault Trees”, in *Proceedings of the 4th International Computer Performance and Dependability Symposium, IPDS '00*, pp. 35–, IEEE Computer Society, 2000.
- [Paderborn12] U. Paderborn, “FUJABA Tool Suite”, 2012. Online. Accessed on 06/10/2014. Available at: http://www.fujaba.de/no_cache/home.html.
- [Paige08a] R. Paige, L. Rose, X. Ge, D. Kolovos, and P. Brooke, “FPTC: Automated safety analysis for Domain-Specific languages”, in *MoDELS Workshops '08*, vol. 5421, pp. 229–242, 2008.
- [Paige08b] R. F. Paige, L. M. Rose, X. Ge, D. S. Kolovos, and P. J.

- Brooke, “FPTC: Automated Safety Analysis for Domain-Specific Languages.”, in M. R. V. Chaudron, editor, *MoDELS Workshops*, vol. 5421 of *Lecture Notes in Computer Science*, pp. 229–242, Springer, 2008.
- [Papadopoulos11] Y. Papadopoulos, M. Walker, D. Parker, E. Rude, R. Hamann, A. Uhlig, U. Gratz, and R. Lien, “Engineering failure analysis and design optimisation with HiP-HOPS”, *Engineering Failure Analysis*, vol. 18, no. 2, 590–608, 2011.
- [Pardo-Castellote03] G. Pardo-Castellote, “OMG Data-Distribution Service: Architectural Overview”, in *Int. Conference on Distributed Computing Systems Workshops*, vol. 0, pp. 200–206, IEEE, 2003.
- [Perez14] D. Perez, R. Mirandola, and J. Merseguer, “On the Relationships between QoS and Software Adaptability at the Architectural Level”, *Journal of Systems and Software*, vol. 87, 17, 2014.
- [Peterson81] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.
- [Portinale10] L. Portinale, D. C. Raiteri, and S. Montani, “Supporting reliability engineers in exploiting the power of Dynamic Bayesian Networks”, *International Journal of Approximate Reasoning*, vol. 51, no. 2, 179 – 195, 2010.
- [Powell95] D. Powell, “Failure Mode Assumptions and Assumption Coverage”, Tech. rep., 1995.
- [Price02] C. Price and N. Taylor, “Automated multiple failure FMEA”, *Reliability Eng. & System Safety*, vol. 76, 1–10, 2002.
- [Priesterjahn11a] C. Priesterjahn, C. Sondermann-Wolke, M. Tichy, and C. Holscher, “Component-based hazard analysis for mechatronic systems”, in *Proc. of ISORCW’11*, pp. 80–87, 2011.
- [Priesterjahn11b] C. Priesterjahn, D. Steenken, and M. Tichy, “Component-based timed hazard analysis of self-healing systems”, in *Pro-*

ceedings of the 8th workshop on Assurances for self-adaptive systems, ASAS '11, pp. 34–43, ACM, 2011.

- [Puliafito14] A. Puliafito, “Web SPN Tool”, 2014. Online. Accessed on 06/10/2014. Available at: http://mdslab.unime.it/webspn/home_page.htm.
- [Pullum01] L. L. Pullum, *Software Fault Tolerance Techniques and Implementation*, Artech House, Inc., Norwood, MA, USA, 2001.
- [Raiteri11] D. C. Raiteri, “Integrating several formalisms in order to increase Fault Trees’ modeling power”, *Reliability Engineering & System Safety*, vol. 96, no. 5, 534–544, 2011.
- [Rao09] Rao, K. Durga, V. Gopika, V. V. S. Sanyasi Rao, H. S. Kushwaha, A. K. Verma, and A. Srividya, “Dynamic fault tree analysis using Monte Carlo simulation in probabilistic safety assessment”, *Reliability Eng. and System Safety*, vol. 94, no. 4, 872–883, 2009.
- [Rausand03] M. Rausand and A. Høyland, *System Reliability Theory: Models, Statistical Methods and Applications Second Edition*, Wiley-Interscience, 2003.
- [Rawashdeh06] O. Rawashdeh and J. Lumpp Jr., “Run-time behavior of Ardea: A dynamically reconfigurable distributed embedded control architecture”, in *IEEE Aerospace Conference Proceedings*, 2006.
- [Riedl08] M. Riedl, J. Schuster, and M. Siegle, “Recent Extensions to the Stochastic Process Algebra Tool CASPA”, in *Proceedings of the 2008 Fifth International Conference on Quantitative Evaluation of Systems*, QEST '08, pp. 113–114, IEEE Computer Society, Washington, DC, USA, 2008.
- [Riedl12] M. Riedl and M. Siegle, “A Language for REconfigurable dependable Systems: Semantics & Dependability Model Transformation”, in *Proc. 6th International Workshop on Verifica-*

- tion and Evaluation of Computer and Communication Systems (VECOS'12)*, pp. 78–89, British Computer Society, 2012.
- [Robidoux10] R. Robidoux, H. Xu, L. Xing, and M. Zhou, “Automated Modeling of Dynamic Reliability Block Diagrams Using Colored Petri Nets”, *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, vol. 40, no. 2, 337–351, 2010.
- [Romain07] B. Romain, J.-J. Aubert, P. Bieber, C. Merlini, and S. Metge, “Experiments in model based safety analysis: Flight controls”, in *DCDS'07*, pp. 43–48, 2007.
- [Rugina07] A. Rugina, K. Kanoun, and M. Kaâniche, “A system dependability modeling framework using AADL and GSPNs”, in *Architecting dependable systems IV, LNCS*, vol. 4615, pp. 14–38, Springer, 2007.
- [Sanders02a] W. H. Sanders and J. F. Meyer, “Lectures on Formal Methods and Performance Analysis”, chap. Stochastic Activity Networks: Formal Definitions and Concepts, pp. 315–343, Springer-Verlag New York, Inc., New York, NY, USA, 2002.
- [Sanders02b] W. H. Sanders and J. F. Meyer, “Lectures on Formal Methods and Performance Analysis”, chap. Stochastic Activity Networks: Formal Definitions and Concepts, pp. 315–343, Springer-Verlag New York, Inc., New York, NY, USA, 2002.
- [Sanders12] W. Sanders, “Mobius user manual”, *Version 2.4.1, University of Illinois*, 2012.
- [Shannon38] C. E. Shannon, “A Symbolic Analysis of Relay and Switching Circuits”, *Transactions of the AIEE*, vol. 57, 713–723, 1938.
- [Sharma11] V. K. Sharma, M. Agarwal, and K. Sen, “Reliability evaluation and optimal design in heterogeneous multi-state series-parallel systems”, *Information Sciences*, vol. 181, no. 2, 362 – 378, 2011.

- [Shelton04] C. P. Shelton and P. Koopman, “Improving System Dependability with Functional Alternatives”, in *Proc. of DSN’04*, pp. 295–304, IEEE, 2004.
- [Signoret13] J.-P. Signoret, Y. Dutuit, P.-J. Cacheux, C. Folleau, S. Collass, and P. Thomas, “Make your Petri nets understandable: Reliability block diagrams driven Petri nets”, *Reliability Engineering & System Safety*, vol. 113, 61 – 75, 2013.
- [SINTEF09] SINTEF, “Offshore Reliability Data Handbook”, Tech. rep., 2009.
- [Somani97] A. K. Somani and N. H. Vaidya, “Understanding Fault Tolerance and Reliability”, *Computer*, vol. 30, no. 4, 45–50, Apr. 1997.
- [Staroswiecki89] M. Staroswiecki and P. Declerck, “Analytical Redundancy in Non-linear Interconnected Systems by means of Structural Analysis”, in *Proceedings of IFAC/IMACS/IFORS Conf. AIPAC’ 89*, vol. 2, pp. 23–27, Elsevier, Nancy, France, 1989.
- [Staroswiecki99] M. Staroswiecki, S. Attouche, and M. L. Assas, “A Graphic Approach for Reconfigurability Analysis”, in *Proc. DX’99*, Jun. 1999.
- [Steiner12] M. Steiner, P. Keller, and P. Liggesmeyer, “Modeling the Effects of Software on Safety and Reliability in Complex Embedded Systems”, in *Computer Safety, Reliability, and Security*, vol. 7613, pp. 454–465, Springer Berlin Heidelberg, 2012.
- [Strigini05] L. Strigini, “Fault Tolerance Against Design Faults”, in H. Diab and A. Zomaya, editors, *Dependable Computing Systems: Paradigms, Performance Issues, and Applications*, pp. 213–241, John Wiley & Sons, 2005.
- [Svard10] C. Svard and M. Nyberg, “Residual Generators for Fault Diagnosis Using Computation Sequences With Mixed Causality Applied to Automotive Systems”, *Systems, Man and Cyber-*

- netics, Part A: Systems and Humans, IEEE Transactions on*, vol. 40, no. 6, 1310–1328, Nov 2010.
- [Tang04] Z. Tang and J. Bechta Dugan, “Minimal cut set/sequence generation for dynamic fault trees”, in *Reliability and Maintainability, 2004 Annual Symposium - RAMS*, pp. 207–213, Jan 2004.
- [Thuel94] S. Thuel and J. Strosnider, “Enhancing Fault Tolerance of Real-Time Systems through Time Redundancy”, in G. Koob and C. Lau, editors, *Foundations of Dependable Computing*, vol. 285 of *The Kluwer International Series in Engineering and Computer Science*, pp. 265–318, Springer US, 1994.
- [Trapp07] M. Trapp, R. Adler, M. Forster, and J. Junger, “Runtime adaptation in safety-critical automotive systems”, in *Proc. of International Conference on Software Engineering*, 2007.
- [Trivedi02] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing and Computer Science Applications*, John Wiley and Sons Ltd., Chichester, UK, 2nd edition edn., 2002.
- [TU Berlin07] R.-T. S. TU Berlin and R. group, “TimeNET 4.0”, 2007. Online. Accessed on 06/10/2014. Available at: <http://tu-ilmenu.de/TimeNET>.
- [TU Kaiserslautern09] A. S. TU Kaiserslautern and F. IESE, “Embedded system safety and reliability analyzer (ESSaRel)”, 2009. Online. Accessed on 06/10/2014. Available at: <http://essarel.de>.
- [Twente14] U. Twente, “DFT Calc”, 2014. Online. Accessed on 06/10/2014. Available at: <http://fmt.ewi.utwente.nl/tools/dftcalc/>.
- [Valmari98] A. Valmari, “The state explosion problem”, in *Lectures on Petri nets I: Basic models*, pp. 429–528, Springer, 1998.
- [vanderBorst01] M. van der Borst and H. Schoonakker, “An overview of PSA

- importance measures.”, *Reliability Engineering and System Safety*, vol. 72, no. 3, 241–245, 2001.
- [Vesely02] W. Vesely, J. Dugan, J. Fragola, Minarick, and J. Railsback, “Fault Tree Handbook with Aerospace Applications”, Handbook, NASA, 2002.
- [Vinod08] G. Vinod, T. Santosh, R. Saraf, and A. Ghosh, “Integrating Safety Critical Software System in Probabilistic Safety Assessment”, *Nuclear Engineering and Design*, vol. 238, no. 9, 2392 – 2399, 2008.
- [Virginia03] U. Virginia, “Galileo”, 2003. Online. Accessed on 06/10/2014. Available at: <http://http://www.cs.virginia.edu/~ftree/>.
- [Walker09] M. Walker and Y. Papadopoulos, “Qualitative temporal analysis: Towards a full implementation of the Fault Tree Handbook”, *Control Eng. Practice*, vol. 17, no. 10, 1115–1125, 2009.
- [Walter08] M. Walter, M. Siegle, and A. Bode, “OpenSESAME: the simple but extensive, structured availability modeling environment”, *Reliability Engineering & System Safety*, vol. 93, no. 6, 857 – 873, 2008.
- [Walter09] M. Walter, “OpenSESAME - Simple but Extensive Structured Availability Modeling Environment”, 2009. Online. Accessed on 06/10/2014. Available at: <http://www.lrr.in.tum.de/~walterm/opensesame/>.
- [Wang04] W. Wang, J. Loman, and P. Vassiliou, “Reliability importance of components in a complex system”, in *Reliability and Maintainability, 2004 Annual Symposium - RAMS*, pp. 6–11, 2004.
- [Wang08] F.-Y. Wang and D. Liu, *Networked control systems*, Springer, 2008.
- [Wilfredo00] T. Wilfredo, “Software Fault Tolerance: A Tutorial”, Tech. rep., 2000.

- [Wolforth10] I. Wolforth, M. Walker, L. Grunske, and Y. Papadopoulos, “Generalizable safety annotations for specification of failure patterns”, *Softw. Pract. Exper.*, vol. 40, 453–483, 2010.
- [Workshop] G. Workshop, “BStoK Module”, 2014. Online. Accessed on 06/10/2014. Available at: <http://grif-workshop.com/grif/bstok-module/>.
- [Wysocki04] J. Wysocki, R. Debouk, and K. Nouri, “Shared redundancy as a means of producing reliable mission critical systems”, in *Proc. of RAMS’04*, pp. 376 – 381, 2004.
- [Wysocki07] J. Wysocki and R. Debouk, “Methodology for Assessing Safety-critical Systems”, *Int. Journal of Modeling and Simulations*, vol. 27, no. 2, 99–106, 2007.
- [Xie04] M. Xie, Y.-S. Dai, and K.-L. Poh, *Computing Systems Reliability: Models and Analysis*, Springer, 2004.
- [yangLi10] C. yang Li, X. Chen, X. shan Yi, and J. yong Tao, “Heterogeneous redundancy optimization for multi-state series?parallel systems subject to common cause failures”, *Reliability Engineering & System Safety*, vol. 95, no. 3, 202 – 207, 2010.
- [Zio13] E. Zio, *The Monte Carlo Simulation Method for System Reliability and Risk Analysis*, Springer Series in Reliability Engineering, 2013.

Acronyms

AADL Architecture Analysis and Design Language. 44–46

ACC Air Conditioning Control. 15

BDMP Boolean logic Driven Markov Process. 39

CAN Controller Area Network. 201, 204

CCU/BA Control and Communication Unit - Bus Administrator. 200–202

CDF Cumulative probability Distribution Function. 103, 153

CDFT Component Dynamic Fault Tree. 109–113, 115–117, 147

CFP Compositional Failure Propagation. 40, 41, 43, 46

CFT Component Fault Tree. 41–43, 45, 55, 59

CTMC Continuous Time Markov Chain. 35, 36, 38, 39

D3H2 . 65, 66, 76, 81, 97, 147, 149–151, 197, 198

DBN Dynamic Bayesian Networks. 36

DEM Dependability Evaluation Modelling. 99, 100, 102, 104, 108, 120, 123, 150, 152, 154, 162, 195, 196

DFT Dynamic Fault Tree. 34–39, 42, 43, 109

DOE Design of Experiments. 53

DSC Door Status Control. 188

DSPN Deterministic and Stochastic Petri Nets. 39, 235

EFMA Extended Functional Modelling Approach. 69, 77

FCI Failure Criticality Index. 114, 115, 117, 143

FD Fault Detection. 77

FMA Functional Modelling Approach. 68, 69

FMEA Failure Mode and Effect Analysis. 41, 46, 51

FP Fire Protection. 16

FPTN Failure Propagation and Transformation Notation. 41–43

FT Fault Tree. 32, 34, 39–41, 45

FTA Fault Tree Analysis. 51, 53

FTP File Transfer Protocol. 202

GSPN Generalized Stochastic Petri Nets. 36, 38, 45, 52, 235

HiP-HOPS Hierarchically Performed Hazard Origin and Propagation Studies. 41–43, 45, 51, 109

IFT Improved Multi Fault Tree. 51

MF Main Function. 69, 82, 84, 85, 87–89, 91, 93, 95, 162

MIM Multi Interface board Module. 200, 201

MVB Multi-function Vehicle Bus. 198, 201, 202, 204, 207, 208

NCSs Networked Control Systems. 2, 3, 5, 62

NMR N modular redundancy. 26

OCI Operational Criticality Index. 115

PIS Passenger Information System. 17

PL Physical Location. 69, 82, 84, 85, 87–89, 91, 93, 95

PN Petri Nets. 53

PU Processing Unit. 2, 4, 13, 15, 30, 63, 78–80, 83, 84, 86, 90, 94, 96, 100, 107, 125, 128, 129, 135, 139, 140, 143, 157, 174, 183, 198, 204, 206–209, 215

R Reconfiguration. 77

RBD Reliability Block Diagram. 32, 34, 38

RCI Restore Criticality Index. 115

SAN Stochastic Activity Networks. 157–163, 165, 175, 177, 182, 188, 235

SEFT State-Event Fault Tree. 39, 40, 43

SF Subfunction. 70, 82, 84, 85, 87–89, 91, 93, 95, 102, 104, 105, 107, 122, 153–157, 162, 167, 168

SPN Stochastic Petri Nets. 235

TCMS Train Control Monitoring System. 13, 14

TCN Train Communication Network. 198, 199

TFT Temporal Fault Tree. 42, 43

TICO Communication Interface Card. 200–202

TMR Triple Modular Redundancy. 26

UDP User Datagram Protocol. 206, 208

UML Unified Modelling Language. 44, 45