School of Computing

Engineering Holistic Fault Tolerance

Rem Gensh

Submitted for the degree of Doctor of Philosophy in the School of Computing, Newcastle University

September 2018

Abstract

Fault-tolerant software should be engineered to be maintainable as well as efficient with regards to performance and resources. These characteristics should be evaluated before deployment of the software. However, the main focus is very often made on the functional features of the application, whereas fault tolerance mechanisms are neglected. As a result, they are often neither maintainable nor efficient. The concept of Holistic Fault Tolerance was introduced to deal with these issues. It is a novel crosscutting approach to the design and implementation of fault tolerance mechanisms for developing reliable software applications that meet non-functional requirements, such as performance and resource utilisation.

The thesis starts with the description of problems that were motivating for the idea of Holistic Fault Tolerance. These problems are related to resource utilisation requirements of modern computer-based systems, since more resources like hardware components and energy are required to process modern computational tasks and ensure performance and reliability of the computation. Moreover, the complexity of these systems grows, leading to maintainability deterioration, especially of those system parts, which are responsible for satisfying non-functional requirements, such as reliability, performance and resource usage.

After analysis of the problems and motivations, the engineering approach to Holistic Fault Tolerance is introduced and main engineering steps are defined. Next, an architectural pattern for Holistic Fault Tolerance is presented. The method to refine the proposed architecture and ensure efficiency of a particular system under development is demonstrated during the modelling step. Then the implementation of Holistic Fault Tolerance based on the proposed architecture and modelling is described in detail.

Finally, the Holistic Fault Tolerance architecture is evaluated with regards to efficiency and maintainability. The evaluation demonstrates that Holistic Fault Tolerance assists in meeting the non-functional requirements, makes fault tolerance mechanisms easier to maintain and ensures higher modularity of the source code.

Declaration

I declare that no part of the thesis has been previously submitted for a degree or any other qualification at Newcastle University or any other University.

Rem Gensh

September 2018

Publications

Parts of the work within this thesis have been presented in the following publications:

CONFERENCE

1. **R. Gensh**, A. Rafiev, A. Garcia, F. Xia, A. Romanovsky, and A. Yakovlev. Architecting Holistic Fault Tolerance. In: *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*, pp. 5-8. (2017)

2. **R. Gensh**, A. Romanovsky and A. Yakovlev. On structuring Holistic Fault Tolerance. In: *Proceedings of the 15th International Conference on Modularity.* ACM, New York, USA, 2016, pp. 130-133.

WORKSHOP

1. **R. Gensh**, A. Romanovsky and A. Yakovlev. Engineering Cross-Layer Fault Tolerance in Many-Core Systems. In: *Proceedings of 7th International Workshop on Software Engineering for Resilient Systems (SERENE 2015)*, LNCS-9274, 2015.

2. **R. Gensh**, A. Rafiev, F. Xia, A. Romanovsky, and A. Yakovlev. Modelling for Systems with Holistic Fault Tolerance. In: *Proceedings of 9th International Workshop on Software Engineering for Resilient Systems (SERENE 2017)*, LNCS-10479, 2017.

TECHNICAL REPORT

1. **R. Gensh**, A. Garcia, and A. Romanovsky: Experience Report: Evaluation of Holistic Fault Tolerance. School of Computing Science Technical Report Series. School of Computing Science, Newcastle University (2017)

Acknowledgements

First of all, I would like to thank my supervisor Alexander Romanovsky for countless support during this challenging and prolonged PhD journey. He introduced me to the area of dependability and fault tolerance and has been mentoring me throughout the studies. Without his inspiration and guidance, I would have never carried out this complex task.

My sincere thanks to my supervisory team Alex Yakovlev and Alexei Iliasov for their motivation and support. Alexei provided very good advice in finding the path of my research during the initial steps of my PhD study.

Special thanks to Ashur Rafiev, Fei Xia and Rishad Shafik who helped me to devise and develop the idea of Holistic Fault Tolerance. We spent a lot of time discussing it, and finally I've got a coherent story. Collaboration with Fei and Ashur was very productive for Holistic Fault Tolerance modelling support.

I am thankful to Anatoliy Gorbenko for fruitful discussions about the similarities and differences of my ideas with the concepts of the TCP/IP protocol. Eventually, the TCP/IP example became one of the main motivations for Holistic Fault Tolerance.

I am profoundly grateful to Alessandro Garcia who advised me to apply Aspect-Oriented Programming to evaluate and show the practical use of Holistic Fault Tolerance. I would like to thank him and his team from PUC-Rio University for warm reception during my visits.

I want to thank all members of the PRiME project team with whom I have had the honour to work. These collaborations were very inspirational for my research.

I would like to express my gratitude to Lyudmila Romanovskaya for her support with proofreading of the thesis.

Also, I want to thank my friend Ilya Lopatkin who induced me to try myself in academia and undertake this challenge.

Last but not least, I would like to thank my parents Bruno and Olga for their encouragement during this endeavour. Even though they are far, they have always supported me. Special thanks to my wife Victoria for her patience throughout the study.

Contents

List of Figures

List of Tables

Acronyms

FT Fault Tolerance

HFT Holistic Fault Tolerance

AOP Aspect-Oriented Programming

OOP Object-Oriented Programming

CLFT Cross-Layer Fault Tolerance

ICT Information and Communication Technologies

TCP Transmission Control Protocol

TCP/IP Transmission Control Protocol/Internet Protocol

IP Internet Protocol

UDP User Datagram Protocol

CRC Cyclic Redundancy Check

LLC Logical Link Control

ACID Atomicity, Consistency, Isolation, Durability

WSNs Wireless Sensor Networks

IFTC Idealised Fault Tolerant Component

IMA Integrated Modular Avionics

ADLs Architecture description languages

UML Unified Modeling Language

SANs Stochastic Activity Networks

SAN Stochastic Activity Network

OGs Order Graphs

OG Order Graph

IIP Initial Image Processing

OCR Optical Character Recognition

GUI Graphical User Interface

NPQ Number Plates Queue

LoCs Lines of Code

1

Introduction

Contents

Computer-based systems should provide the expected service according to their specification. The ability to provide such service that can defensibly be trusted, while the service failures are not more frequent than acceptable is known as system *dependability* [1]. The system dependability implies that the system is ready to provide a correct service without interruptions, there are no disastrous consequences on the people and the environment because of system operation, and the system is repairable and modifiable. Faults, errors and failures are the threats to system dependability. Special means are applied to deal with these threats in order to achieve system dependability by preventing, tolerating, removing and forecasting possible faults. *Fault tolerance* is one of dependability means that prevents system failures, even though faults may be present in the system. *Exception* or exceptional condition [2] is an anomalous behaviour when the normal computation flow is changed by the exceptional flow. *Exception handling* [2, 3] is a mechanism of responding to an exceptional condition. It is partially related to fault tolerance. Cristian [4] distinguishes programmed exception handling and default exception handling. The former is a fault avoidance technique, whereas the latter is a fault tolerance technique. System fault tolerance should be ensured during all stages of the system development life cycle.

It is not an easy task to design a dependable system that delivers critical services [5]. The complexity of modern computer-based systems and the diversity of the fault classes require complex Fault Tolerance (FT) mechanisms to ensure system dependability. FT engineering is a crucial part of the design and implementation of computer-based systems. Poor FT engineering decisions could lead to disastrous consequences. There are a lot of examples when badly designed fault tolerance led to very serious or even catastrophic consequences.

The Therac-25 is a radiation therapy machine that was involved in several accidents of massive radiation overdoses that were hundreds of times greater than normal [6]. Three patients died and several were seriously injured after the overdoses. Two software faults caused the accidents. The first fault was activated when the machine operator unintentionally triggered a race condition so that some parameters were initialised incorrectly and did not correspond to the chosen operation mode. The second fault allowed the function to be activated in operation mode that did not support this function. The previous hardware platform had special interlocks to prevent such faults, however, the Therac-25 relied on software checks only. The commission of inquiry concluded the bad software design and development practices were the main cause of accidents [7]. The software was

designed in such a way that it was almost impossible to apply automated tests and check all possible scenarios. Another reason is confusing reliability with safety. The software was highly reliable, since there were only a few failures per tens of thousands of treatment cases. However, this software was not safe because the failures were fatal. Thus, the system was not dependable. The software lacked FT mechanisms that would have detected and properly handled system state inconsistencies and bugs in the source code.

The Northeast Blackout of 2003 was a huge power outage that affected parts of the United States and Canada on August 14, 2003 [8]. The blackout indirectly caused about 100 casualties and resulted in several billion dollar losses in the USA and Canada. There are several causes of this major accident. Badly designed FT mechanisms and ineffective organisational practices significantly contributed to it. Independent review was not conducted by reliability council allowing responsible company to use inadequate development and monitoring approaches. That includes ineffective communication procedures, lack of comprehensive system assessments, especially for extreme conditions, lack of system rigorous planning studies, lack of functional state evaluation of monitoring tools after repairs were made, absence of spare monitoring tools for the case when primary monitoring systems failed. Requirements and standards were not rigorous and they could be interpreted ambiguously, which is insufficient for reliable system operation. Reliability coordinators did not use real-time data to support real-time monitoring. The control room alarm system was stalled for over an hour due to a race condition in the energy management system. This malfunction caused important alerts for system state changes not to be transmitted to system operators.

The launch of the Ariane 5 rocket (flight 501) failed 37 seconds after the start incurring a loss of more than 370 million US dollars [9]. The failure was caused by a hardware exception that was triggered by arithmetic overflow due to conversion from a 64-bit floating point number to a 16-bit signed integer value. The software code with this unprotected conversion was reused from the earlier Ariane 4 version where the bug remained dormant. This operation did not have an exception handler because of the assumption that the variable presented by horizontal velocity is physically limited, whereas other operations were protected by exception handlers.

In many computer-based systems the lack of effort to ensure systems dependability and efficiency is often due to time and budget limits. Even if a computer-based system is developed in accordance with best practice, there is always a trade-off between reliability,

maintainability and efficiency to be considered.

This chapter introduces the research problem investigated in the PhD study. Firstly, motivations for the research are briefly overviewed. Then, the concepts which will be referred to throughout the thesis are discussed. Next, the resources such as energy, hardware components and time, that are required for the functioning of modern computer-based systems are considered. After that, the research question is formulated and the contributions of the thesis are listed. Finally, the structure of the thesis is outlined.

1.1 Growing complexity of fault tolerance

The amount of software is growing drastically in all areas where software is applied, desktop applications, mobile applications, Internet platforms, cloud services, automotive and aerospace industries [10–13]. In the automotive industry the amount of software code demonstrates an exponential growth [14, 15]. In the late 1970s the majority of cars did not have any software, whereas in the late 2000s many premium cars had tens of millions of lines of code. It is inevitable that complexity of these systems is growing as well.

The growing complexity of software leads to a rise in complexity of FT mechanisms in this software. Cristian states [16] that about 70% of code in operational computer software systems is devoted to exception detection and exception handling. In addition, it is claimed that most of the design faults are located in exception handling code, which is the least documented, tested and understood part of the system. The issue of increasing complexity of FT is addressed in [17]. It is shown that the situation in FT design is deteriorating as a result of growing complexity of software in general, and of FT mechanisms in particular.

It is stated in [18] that complexity of the software and accelerated development schedules makes defect avoidance difficult. The statistic is provided that finding and fixing the problem after implementation is 100 times more expensive than during the requirements and design phases. Software projects spend almost half of the effort on avoidable rework. This rework could have been entirely avoided or fixed less expensively. Only about 10% of defects cause about 90% of downtime events. Defect introduction rate can be reduced up to 75% by using disciplined personal practices for the software development process.

A major problem with FT in modern systems is that due to weak documentation and testing it is the least well-understood part of the system. Very often the design of FT

mechanisms is seen as a lower priority and is less systematised than that of functional requirements of computer-based systems. As pointed out in [17], the reasons for this are difficulty of development and support of FT means, additional costs of FT development as a result of applying redundancy, as well as many developers preferring to focus on system functionality rather than pay attention to faults end errors.

1.2 Misuse of fault tolerance techniques

The FT techniques are often misused due to complexity of FT, low priority of FT mechanisms in the development process and not thought-out fault assumptions. The problem of FT mechanism misuse due to system complexity is addressed in [19], where the authors offer special patterns that support developers in rigorous system design. In addition, it is acknowledged that many systems are prone to faults introduced during the early development phases between requirements elicitation and system implementation.

The analysis of commercial-off-the-shelf components [20] has shown that the majority of exceptions, which can be thrown during the code execution, are not documented. The percentage of undocumented exceptions is over 98% in 3 .Net applications, over 90% in 3 .Net libraries and over 80% in 2 .Net infrastructure assemblies.

Oliveira et al. [21] analysed changes of functional and exception handling code in more that 100 versions of 16 Android projects. The analysis has shown that increased usage of Android abstractions is not followed by increased usage of exception handling. As a result, the number of uncaught exceptions that crash the application increases making the application less robust. A team of scientists [22] discovered a previously unknown *uncaughtException* flaw that is caused by careless implementation of exception handling code. In addition, this flaw could even be exploited to undertake a denial of service attack. Maji et al. [23] conducted an empirical study to investigate the robustness of inter-component communication in Android. It was found that many Android components have poor exception handling code. It is shown in [24] that 19% of Android application crashes could have been caused by undocumented exceptions, which are present in 18% of non-private Android API methods. A study analysing the source code and the byte code related to 6005 exception stack traces from 482 projects hosted in GitHub and 157 projects hosted in Google code was conducted in [25]. The outcomes have shown that about 50% of all application crashes happen because of errors in programming logic and

especially due to null pointer exceptions. Other bug hazards include very poor documentation for exceptions thrown by Android platform and third party libraries, extensive use of exception wrappers that deteriorates understanding of exception chains and violates exception handling principles. All of these negatively impact application robustness. Kery et al. [26] argue that Java try/catch exception handling mechanism is flawed. To prove this it was pointed out that the majority of error handlers are of low quality. In this study over 11 million try/catch blocks from GitHub projects were analysed. The examination has shown that much of the time exceptions are handled locally and are not propagated through the call chain. Developers apply bad practices and application-wide implications of the caught exception are not usually considered. The use of actions like Log, Print, Return is widespread. Very often the code is copied between the handlers. Empty catch blocks and universal catch blocks for all types of exceptions are common. In many cases developers prefer not to use exceptions themselves to signal an exceptional condition and suppress exceptions that were caught without attempts to handle them properly. Among the investigated try/catch blocks 12.4% were completely empty, 10% just printed the stack trace, another 10% wrote to a log file only, 5% had a return statement. The majority of actions in 98% of all catch blocks were presented by a single statement like return, print log, method call or throw (when exception is just re-thrown without proper handling). 9 android applications were examined in [27]. The researchers were looking specifically into Android activities – one of the application building blocks on the Android platform. It was demonstrated that more than half of the activities did not have exception handlers. Thus, any exception raised in such activity would cause a crash of the application.

These examples illustrate that the software development process suffers from exception handling misuse because of unsound development practices, complexity of exception handling mechanisms and weak documentation. Eventually, these problems deteriorate maintainability of FT.

1.3 Maintainability of fault tolerance

Software complexity affects software maintainability involving more time, higher costs and more errors introduced after maintenance works [28]. Thus the complexity of software and especially the complexity of FT mechanisms leads to difficulties in maintaining FT in computer-based systems. The problem gets worse due to misuse of FT mechanisms. And yet the FT means comprise a significant part of the application source code. The

experiments in [29] showed that about 11% of the code in the JWAM framework was devoted to detecting and handling exception behaviour, given the fact that the majority of handling statements did not have complex logic for dealing with errors. The JWAM framework is an object-oriented framework for interactive business applications consisting of 614 classes and 44000 lines of code.

It was mentioned in the previous section that there were problems with FT design in the past and such problems still remain in the present. It is well-known that FT is a crosscutting concern [30, 31]. In this way, during the design and implementation of FT functionality the main focus should be made system-wide, rather than on components. It is more convenient to centralise FT-related code in order to facilitate modularity of the system, since the relevant FT functionality will be coordinated by a single module, unit or component, simplifying the understanding and access to the FT mechanisms. However, FT mechanisms are often implemented separately for system components without consideration of the entire system.

A study [32] identifies several problems with exception handling. The most likely explanation is that exception handling, especially in the fast-paced business environment, is not a developers' primary concern, whereas functional features receive much more attention in requirements, design, and testing. In addition, it is extremely hard to predict all possible errors that might emerge during the system operation, and unit tests [33] cannot cover all possible scenarios. Moreover, exception handling is a crosscutting concern, which affects its understandability and maintainability. Finally, a lot of code is written and maintained in programming languages that do not explicitly support exception handling.

The need to consider exception handling during all the phases of software development is explained in [34]. It is demonstrated that systems can be affected by the inappropriate use of error recovery mechanisms due to dealing with exception handling only at later stages of the software development life cycle.

Software maintainability [35] is central in reducing maintenance costs and decreasing downtime in case of system modification or, worse, in case of system failure. However, it often happens that due to system complexity maintenance actions unintentionally introduce new bugs and faults affecting working parts of the computer system. To avoid or at least to minimise these risks, system modules should not be made heavily dependent on each other. In addition, each module should be responsible for certain functionality in such a way that similar operations are not scattered across the system. This good prac-

tice is usually followed while implementing functional system features such as business logic or data access. The situation with non-functional features is, however, different [36]. The source code responsible for diagnostics, security or FT is often distributed across the system, leading to code duplication or tangling with the code responsible for functional concerns. In many cases FT functionality is not centralised and each module performs error handling and fault handling independently, even though the errors involve the entire system. This makes FT mechanisms more difficult to understand, and their adjustment and modification more time-consuming, and ultimately does not support system modularity. Good maintainability, by contrast, means that any modification, be it repairs or adding a new functionality would require an anticipated amount of time and effort.

The above-mentioned problems with FT are related to the development process. Very often developers focus on functional system requirements, whereas non-functional requirements are neglected. Some development teams prefer testing to reveal faults and errors rather than rigorous development. However, this approach is not acceptable for critical systems, such as avionics, medicine and the financial sector [37]. Since the FT code is critical and could contain bugs and design flaws itself, it should be designed and implemented very thoroughly.

1.4 Resource and energy requirements of the ICT sector

Information and Communication Technologies (ICT) are ubiquitous in the modern world and new technologies have expanded to almost all spheres of human life. However, the ICT sector consumes huge amounts of energy. It is pointed out in [38] that US data centres consumed 1.5% of the total US energy consumption or 61 billion kWh in 2006. Moreover, according to Mills [39], the entire ICT sector now consumes about 10% of the energy generated in the whole world. In addition, the area of cryptocurrencies is developing very fast. These digital currencies require huge amounts of computer resources – CPUs, GPUs, and even specialised boards like ASICs [40] to create (mine) new tokens (cryptocoins) and run the network. According to the New York Times [41], the daily energy consumption of computers connected to the Bitcoin [42] digital currency network is comparable to that of medium-size countries. Thus, energy consumption and computer resource usage are starting to become an issue for modern computer-based systems. This makes it necessary to develop new approaches to make these systems more efficient with regards to energy

consumption and resource usage.

The majority of modern computer-based systems are multi-core or even many-core, and the number of cores is expected to grow [43]. Large numbers of CPU cores will increase computer systems energy consumption. However, these systems offer plenty of benefits. Firstly, provided that the software code is written correctly, they operate faster. Secondly, systems' reliability increases due to redundancy intrinsic to such systems. Finally, modern multi-core and many-core systems make it possible to adjust performance, CPU energy consumption and even reliability, by changing frequency of CPU cores and switching off unused cores (and running only the required number of cores).

Some studies have demonstrated that it is possible to optimise resource usage if thorough analysis is done at the stage of FT mechanism design. For example, Garraghan et al. [44] quantified the negative influence of failures on energy consumption in a large-scale cloud system. The results show that most failure events happen in low priority tasks causing 13% of total energy waste, whereas failures which happen in high priority tasks cause 8% of total energy waste. This suggests that system-wide analysis is vital in identifying the main factors of failure-related energy waste. Energy waste can be reduced while providing a required quality of service. To achieve this, FT mechanisms should take into account task priority, energy profiles and frequency of failures inside components.

1.5 Holistic Fault Tolerance

Holistic Fault Tolerance (HFT) is introduced to ensure efficient use of resources available to computer-based systems and to address the questions of FT complexity and FT maintainability raised in Section 1.1 and Section 1.3. This thesis proposes the concept of HFT and the engineering support for computer-based systems developed with HFT. The notion of HFT is described in detail and the guidance to support developers during the design and implementation of HFT for computer-based systems is provided.

The concept of HFT assumes that a global, crosscutting approach should be applied to manage system non-functional characteristics at all stages of the software development life cycle. These non-functional characteristics are presented by FT, performance and resource usage. HFT is targeting resources in general, but hardware components, energy and time are more frequently-used. Depending on the system type some resources could be more important than others.

To assist in applying a crosscutting approach, the computer-based system should be developed in accordance with the HFT architecture. This involves introducing a special HFT controller for managing non-functional properties of the systems. To simplify the implementation and understanding, the controller is assisted by the HFT agents, which monitor certain properties of system components and supply processed data to the HFT controller. The goal of the HFT architecture is making it convenient to maintain FT-related mechanisms, and ensuring dependable and efficient operation of a computer-based system. HFT is a general concept and it is suitable for the majority of modern computer-based systems, however multi-core and many-core systems provide more flexibility and benefits for HFT.

1.6 Research question

The research question of this thesis is to examine the effect of HFT on the efficiency and maintainability of computer-based systems. It is necessary to investigate whether HFT can monitor and dynamically adjust the entire computer-based system to achieve efficient operation without uncontrollable reliability deterioration. By efficiency we mean the ability of a computer system to avoid the waste of time or computer resources. We also argue that HFT facilitates the maintainability of FT functionality. To demonstrate this, the thesis evaluates the HFT architecture, both in terms of its efficiency and its maintainability. The former is based on analysing the interplay between reliability, performance and resource utilisation. The latter considers modifications of FT-related functionality.

1.7 Contributions

The four main contributions of the thesis are:

- The concept of Holistic Fault Tolerance. The main idea of HFT and the steps required to design, model, and implement a computer-based system with HFT are presented.

- A detailed description of the HFT architecture, including all elements of the HFT architecture and their interactions.

- A general method intended to facilitate the design and implementation of the HFT architecture. In this work, a modelling method for systems with the HFT architecture is provided.
- The pattern for the implementation of HFT using the AspectJ Aspect-Oriented Programming (AOP) [45] extension for Java programming language.

The proposed method is evaluated with regards to the efficiency and maintainability of the HFT architecture.

1.8 Outline of the thesis

In Chapter 2 we discuss several motivations for this research and provide the background; Chapter 3 is devoted to HFT engineering. The HFT architecture is described in Chapter 4, whereas Chapter 5 focuses on modelling support for computer-based systems with the HFT architecture. The implementation of HFT using AOP is demonstrated in Chapter 6. HFT efficiency and maintainability are evaluated in Chapter 7, and results are discussed and concluding remarks made in Chapter 8.

2

Motivation and background

Contents

2.1 Introduction

Several academic studies and industrial solutions that address various issues relating to the efficiency and maintainable architectures of computer-based systems are considered in this chapter. A critical analysis of these state-of-the-art approaches is provided, highlighting their advantages and limitations, and explaining how these works inspired us with the idea of Holistic Fault Tolerance (HFT) and motivated us to undertake this research project.

This chapter is organised as follows. First, the background on dependability, fault tolerance and fault-tolerant systems is introduced in Section 2.2. Then, the ideas of Cross-Layer Fault Tolerance (CLFT) (Section 2.3) and centralised Fault Tolerance (FT) (Section 2.4) are investigated and the logic behind cross-layer and centralised approaches is explored. After that, the existing approaches to system structuring that do not always follow conventional ways of system structuring are discussed in Section 2.5. Thereupon, a real world Transmission Control Protocol/Internet Protocol (TCP/IP) [46] example motivating the idea of HFT is introduced in Section 2.6. Concluding remarks of the chapter are given in Section 2.7.

2.2 Basic concepts and taxonomy of dependability

The main concepts and taxonomy of dependability are an established scientific theory thoroughly explained in [1]. Dependability is defined as the ability of the system to provide service that can be justifiably trusted. The concept of dependability encompasses five attributes:

- *availability* — readiness of the system to provide a correct service.
- *reliability* — continuity of correct service provided by the system.
- *safety* — absence of disasters caused by the system operation.
- *integrity* — absence of inappropriate changes in the system.
- *maintainability* — ability of modifications and repairs.

2.2.1 Threats to dependability

There are three threats to dependability: faults, errors and failures. A *fault* is a hypothesised cause of an error. If a fault causes an error, it is active. Otherwise it is dormant.

An *error*, in turn may cause a failure if the error reaches the service interface of a system. A *failure* or a service failure, is a certain event when the provided service deviates from correct service, and as a result, the system behaviour does not comply with the system specification. An *external state* of the system is that part of the total system state which is provided by the service interface [1]. The recursive chain of dependability threats is shown in Figure 2.1.

$$\ldots \rightarrow \text{failure} \xrightarrow{causation} \text{fault} \xrightarrow{activation} \text{error} \xrightarrow{propagation} \text{failure} \xrightarrow{causation} \text{fault} \rightarrow \ldots$$

Figure 2.1: Fault, error, failure chain

One of possible examples of faults, errors and failures manifestations is shown in Figure 2.2. There are three components in the system. An activation of an internal fault in

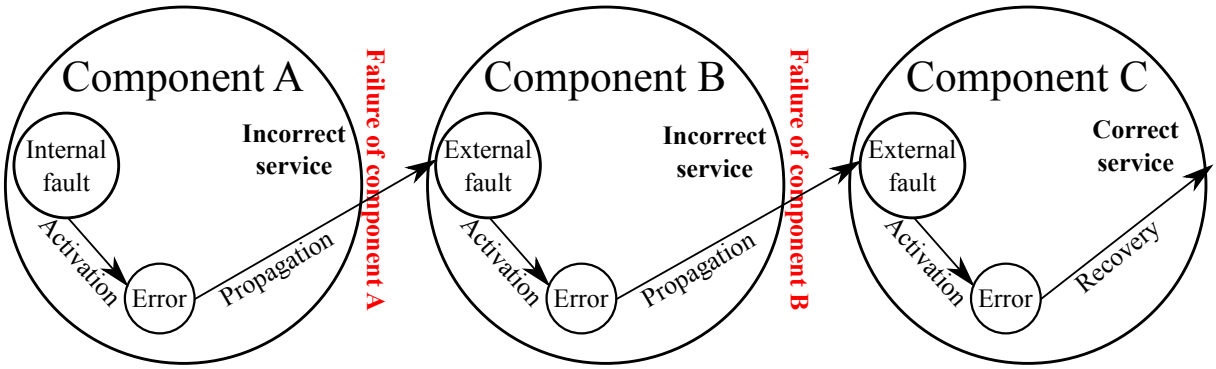

Figure 2.2: Error propagation through components

component A causes an error in this component. When there is no recovery of the error, it is propagated through the component and reaches an external state of the component resulting in a failure of component A. The failure of component A causes an external fault activation in dependent component B. The active fault produces an error in component B that is not recovered (or even not detected) correctly. Thus, again, the error will be propagated to the external state of component B causing deviation of the provided service from the correct service or failure of component B. The failure of component B causes an external fault in component C leading to an error. This error will be detected and recovered by component C. The error recovery stops error propagation through system components and allows component C to return to the normal operation and provide a correct service for the external components. Thereby, the error will not reach the external state of the system.

2.2.2 Types of faults

There is a great variety of faults threatening dependability of computer-based systems. These faults are classified into eight classes [1]:

1. *Phase of creation* class is presented by *development faults* that occur during development or maintenance phases and *operational faults* that occur during the system operation phase.

2. *System boundaries* class distinguishes *internal faults* originating inside the system boundary and *external faults* originating outside the system boundary. The system boundary defines a frontier between the system and its operating environment. The errors caused by external faults are propagated to the system through the interfaces. For example, let us consider the interaction of two components in a computer-based system, a hard disk drive (disk) and a software application. The disk could have an internal fault due to tear and wear which may finally lead to an error. If this error is detected and recovered by the disk itself, it would not reach the external state (or interface) of the disk and it would not be propagated to the application. So the disk error caused by an internal fault will be recovered allowing the disk to provide the correct service to the application in the presence of fault. However, the disk may not be able to recover the error and it would reach the external state of the disk leading to the disk's service failure. This failure would be an external fault for the software application. If the disk operability is not significantly affected and only some parts of data are corrupted (e.g. several bytes out of kilobyte), then the application could recover the errors caused by this external fault. For instance, special data structures with redundancy may help to mask these errors. Thus, in the software application the error could be either recovered or propagated further, leading to the correct service in the former case and the application failure in the latter case.

3. *Phenomenological cause* consists of *human-made faults* resulting from human actions and *natural faults* that are caused by natural phenomena without human involvement.

4. *Dimension* class is composed of *hardware faults* that occur in, or affect system hardware and *software faults* ("bugs" and implementation flaws) that affect program

or data correctness.

5. *Objective* class encompasses *malicious faults* that are introduced with objective to cause harm to the system and *non-malicious faults* that are introduced without malicious objective.

6. *Intent* class distinguishes *deliberate faults* that are introduced by bad decisions and *non-deliberate faults* that are introduced without any harmful intent.

7. *Capability* class comprises *accidental faults* introduced by oversight and *incompetence faults* introduced as a result of incompetence.

8. *Persistence* class includes three categories of faults: permanent, intermittent and transient. Permanent faults are constant, for example stuck-at-0 or stuck-at-1 chip faults. The difference between an intermittent fault and a transient fault is that the former occurs in the same location. In contrast, random bit-flips are inherent for the latter. The main reason for intermittent faults in hardware is device wear out, especially in nanoscale nodes. The study [47] states that current detection and recovery approaches consider only two cases: transient and permanent faults. Shrinking of circuit components also makes hardware susceptible to device variation and ageing effects.

Appropriate error detection, error recovery and fault handling techniques are required for each type of fault to ensure dependable system operation and to decrease FT overheads.

2.2.3 Dependability means

The means to attain the attributes of dependability are grouped into four categories: fault prevention, fault tolerance, fault removal and fault forecasting [1]. *Fault prevention* is applied to prevent the faults from occurrence and introduction in the system. The aim of *fault tolerance* is to avoid system failure while faults are present in the system. Fault tolerance includes identification of error presence—*error detection*, elimination of the error from the system state—*error handling*, and prevention of the fault/s causing the error from future activation—*fault handling*. Error handling is presented by *rollback* (or backward error recovery), *rollforward* (or forward error recovery), and *compensation* mechanisms. Rollback restores an earlier system state, which was not affected by the error. Typical rollback techniques are checkpointing, recovery blocks and system restarts.

Rollforward is suitable for errors that can be entirely removed from the system state allowing the system to continue normal operation. Rollforward is usually considered faster than rollback, however it requires a precise knowledge about the detected error to apply a correct error handler. Rollback and rollforward mechanisms are not mutually exclusive and they complement each other. In case of compensation the error can be masked by using erroneous state redundancy without any other additional redundancy. Various error correction codes are examples of compensation mechanism. Rollback and rollforward are executed on demand after an error is detected, whereas compensation is applied independently of the error detection.

Fault prevention and fault tolerance are responsible for delivering the service that can be justifiably trusted. Fault removal is applied during the two phases—system development and system use—to diminish the number and severity of faults. Fault forecasting is used to estimate current and future faults and to predict possible fault activation order. Fault prevention and fault removal comprise the *fault avoidance means* group, whereas FT and fault removal are part of the *fault acceptance means* group.

2.2.4 Relation between fault tolerance and maintainability

One of the definitions of dependable system is "the ability to avoid service failures that are more frequent and more severe than is acceptable" [1]. Fault tolerance is a means of dependability, which prevents the system failure in the presence of faults. FT consists of error detection, error handling and fault handling. Maintainability along with availability and reliability is an attribute of dependability. Maintainability represents ability of modifications and repairs. Maintenance, in turn, comprises all modifications of the system during the use phase of the system life cycle. There are four forms of maintenance: corrective, preventive, adaptive and augmentative. The first two are related to repairs, whereas the last two are applied for modifications. The goal of corrective maintenance is to remove faults that were isolated by fault handling. The difference between FT and maintenance is that the latter requires an external agent. With regards to the HFT architecture we focus on convenience of FT maintainability and ensuring dependable system operation.

2.2.5 *Ensuring fault tolerance of system components*

It is established [48] that all FT techniques must rely on redundancy for error detection and for error recovery. Moreover, in case of software the redundancy of design is required. Implementation of FT techniques depends on fault assumptions, by defining faults or classes of faults that should be tolerated. The redundancies provided by FT techniques should be independent of the process of fault creation and activation.

Let us look first into several examples of how FT of system components is typically ensured. Triple modular redundancy [49] is a form of N-modular redundancy when three components perform the same operation and a single output is produced by a majority-voting system. The recovery block [50] works with several implementations of the same algorithm: after executing the primary variant, an acceptance test verifies the results. If the acceptance test fails, the system is rolled back and the secondary variant is tried. Eventually, either a variant passes the test or an exception handler is invoked. The N-version programming [51] is an approach aiming to reduce the probability of software faults by developing two or more functionally equivalent program versions independently (preferably by different developers) in accordance with the same initial specification. These versions are executed concurrently and a special voting algorithm choses the correct output.

The two typical approaches used to ensuring FT at several layers are action nesting and extending component interfaces with exceptions. The best examples of the former are exception handling and nested Atomicity, Consistency, Isolation, Durability (ACID) transactions [52, 53]. The latter are best represented by F. Cristian's approach to providing recovery for modular software [54] and the Idealised Fault Tolerant Component (IFTC) pattern [55]. Even though these techniques support layered system structuring for FT they do not support concerted or centralised FT at multiple layers when the decision to apply error detection and recovery is made for all layers together.

Cristian discusses programmed exception handling and default exception handling in [4]. In addition, the usage of default exception handling for the design faults is discussed. It is suggested that in spite of the fact that programmed exception handling and recovery block were developed separately, they should be used in combination, but not as competitive mechanisms. The author noticed that exceptions should not be used for "monitoring purposes". The exception mechanism is a control structure of the programming language, which is applied when an exception is detected. In this case, the standard continuation

of the operation is replaced by exceptional continuation. Cristian claims that algorithmic faults can be the cause of system failures. Backward error recovery is proposed as a mechanism for dealing with these faults. It is not easy to find the golden mean between fault avoidance and FT to cope with design faults. Default exception handling, based on backward error recovery can be an effective mechanism to tolerate design faults when time between manifestation of the fault and detection of the error is less than the time between the beginning and the end of a transaction. Otherwise, this technique will not be adequate.

2.2.6 Focus of the standard approach

The issue with the standard approach to FT structuring (Subsection 2.2.5) is that it often focuses on providing FT and does not give enough attention to other non-functional parameters, such as performance and resource usage. In addition, the FT mechanisms are managed inside the system components, which makes it difficult to reason about the system FT at the system level, rather than at the level of individual components.

When the error is detected, its criticality can be often reasoned about only at the system level, since possible influence of the error can not be predicted at the level of individual system component. In some cases, the system could be significantly overengineered if there is not a centralised control for FT mechanisms. For example, the IFTC pattern can be applied recursively to engineer a system (some of IFTCs may be formed by several idealised fault tolerant subcomponents). While this structure can be applied to create very reliable system, since in IFTC the normal and erroneous operations are clearly separated, such an architecture could be inefficient because the error will be analysed and recovery attempt will be made by each of the components involved in the error propagation chain. At the same time if there is a global view of the system-wide FT mechanisms the decisions about the errors can be made globally without propagating these errors through all components in the call chain.

It seems that the system units (layers or components) are usually considered separately when the standard approach is applied. Under such circumstances, it is likely to be difficult to adjust these units to achieve efficient system operation in terms of performance, energy consumption or resource utilisation. Unnecessary error handlings are possible when the caller component (or upper layer) cannot specify the required quality of service of the callee component (or bottom layer).

For example, let us consider a many-core system where the fault rate of one core is significantly larger than the rate of another core. When an error is detected, the error recovery could be achieved by re-executing the calculations on the same core making it slower. Should there be a special system-wide mechanism, which can make a decision that under some fault rate value, hardware layer error recovery should be applied, but after exceeding this value, it is necessary to inform the Operating System about the faulty core, then the system FT as a whole would be more efficient. In the latter case, the Operating System would be capable to hide the faulty core from the applications for some time allowing it to cool down, because the likely reason of faults is CPU core overheating. In case when all CPU cores became faulty, different strategies could be applied depending on fault assumptions, application domain and the system itself. If the system is just one node of a service consisting of several nodes then the entire node can be switched-off to allow the CPU cores to cool down, while the other nodes will deliver the service. However, if it is a monolithic system then hiding half of the CPU cores can be considered, after the specified time these cores will be made available again and the other half of the CPU cores will be hidden. Another option is to apply graceful degradation of the service.

2.3 Cross-Layer Fault Tolerance

2.3.1 Cross-Layer Reliability Vision

The Cross-Layer Reliability Visioning Study [56] proposes that it is necessary to use a cross-layer, full-system-design approach to reliability. The authors argue that in a cross-layer reliable system the entire system stack needs to collaborate in order to recover the errors and tolerate variations. This will be achieved because the relevant information about the system state is shared across the layers. In addition, the application domain of the system should always be taken into account, since different domains have various reliability requirements.

The study introduces the cross-layer approach to reliable system design, forecasting that the electronics industry is about to approach two inflection points that require new approaches for FT structuring. The first point is reliability and predictability. In the fabrication technologies less than 65nm gate leakage became a serious problem that led to reliability deterioration. This will push the designers to alter the assumptions that semiconductors and other microelectronic elements will operate without fails during the

whole system lifetime. The second point is energy consumption, which is a crucial issue for contemporary computer-based systems.

The Cross-Layer Reliability Visioning Study motivates a new approach to system reliability based on a cross-layer system design. The growing energy demand and the forecasted reliability deterioration of modern hardware ground the importance of the cross-layer approach to reliable system design. CLFT can be applied to cope with these problems. With a cross-layer approach, it is possible to make a decision whether the error at the bottom layer is critical for the upper layers and apply the most suitable error recovery scenario according to the system's state. Thus, CLFT decreases error recovery overheads and reduces energy consumption, because in case of an error an efficient FT strategy will be chosen. CLFT is proposed to be very useful for performance- and energy-critical systems.

2.3.2 Studies on Cross-Layer Fault Tolerance

The CLFT assumes that FT mechanisms are distributed among all layers of the system stack and designed together. The final decision on using FT mechanisms is made according to the whole system state rather than to the states of the individual layers separately.

The *Relax* language-level mechanism for providing energy-efficient reliability and CLFT for supercomputers is proposed in [57]. The Relax framework allows the developer to specify code regions where low-reliability computations should be tolerated. The framework provides the *relax blocks* marking the regions in source code that can experience the error propagated from the hardware layer. This error will be detected and recovery will be performed by re-execution of the *relaxed* code block. Actually, the developer has a choice to retry the operation or skip the operation if it would not significantly affect the calculations. The last option is suitable for the processing of massive data sets when skipping a certain amount of operations can be tolerated (e.g. statistical calculations, digital signal processing). Thus, it can be considered as trading reliability for energy efficiency and performance. This example shows that it is feasible to deal with hardware error at the software layer, given that the error is detected.

The cross-layer design is widespread in the area of Wireless Sensor Networks (WSNs). The wireless sensors [58] are small devices which measure temperature, humidity, air quality, etc. These sensors are networked to each other and transmit their data to the main server.

Very often the battery is the only power supply for them. Therefore, reliability, performance and energy consumption are the most important requirements for these systems. The efficient operation of the WSNs can be guaranteed only when the layers of the system stack are considered together. The layered approach does not provide an easy way to share any important information about the system state among the system layers, this makes it impossible to achieve the efficient system operation. In addition, the single layer approach is incapable of adapting to the environmental change. The paper [59] discusses cross-layer adaptivity techniques, which leverage functionalities at different layers of the protocol stack. The application layer is frequently involved in these activities, supporting current system operation in accordance with measurements and forecasts of the monitored system. The study [60] proposes a new routing protocol based on the cross-layer principle in order to manage faults in WSNs, decrease signalling overhead and power consumption. A cross-layer data delivery protocol for delay/fault-tolerant mobile sensor networks was developed in [61]. The protocol aims to optimise energy consumption in the light of throughput requirement, stable connectivity of the sensor nodes and sufficient channel bandwidth. In spite of the success in applying the cross-layer approach, there is no work on systematic engineering of CLFT in the domain of WSNs. The developers design systems without any support for reuse, often producing similar solutions independently.

The CLFT is a promising approach to the development of efficient systems, however it is more suitable for a layered hierarchy and it would not fit all computer systems. We realised that the idea of CLFT needs to be reworked to be applicable for systems that are built out of components and do not have explicit layered structure. This was one of the reasons for introducing a new holistic approach presented later in the thesis.

2.4 Studies on centralised fault tolerance

The centralisation of FT management is considered in a study [62], which provides the notion of *guardian* – a special global exception handler for a distributed system. In addition, the authors consider the implementation of distributed exception handling and global exception handling and analyse the provided guardian model. The goal of the guardian is to enhance existing exception handling models and provide a basis for them. The authors note that in a distributed system exception handling is far different from sequential exception handling, since distributed systems require communication and coordination of exception handlers. Moreover, several exceptions may be raised concurrently. Thus,

each participating process of the application should invoke the correct handler. In the guardian model, the correct handler for each process is chosen by the guardian according to the application defined recovery rules allowing the guardian to orchestrate the recovery action of each involved process. The guardian model distinguishes global exceptions coordinated through the guardian and local exceptions, which are handled by those processes where they occurred. The authors list three advantages of the guardian model [62]. First, the global exception handling is separated from the participating processes. Second, the guardian is a more flexible and primitive scheme than existing approaches to distributed exception handling. Third, the synchronous and asynchronous activities can interact in a distributed application. However, the implementation of reliable broadcast and time-limited response of participating processes involves scalability and performance overheads. The second limitation is a complexity of definition of the contexts and corresponding set of exceptions and handlers according to the guardian rules.

A three-layer architecture for FT control is introduced in [63]. These layers are control, detection and supervision. The first layer is responsible for controlling sensors and actuators that check faulty conditions. The second level contains detectors for each fault effect and corresponding effectors implementing reconfigurations and remedial actions initiated by an autonomous supervisor from the third layer. To achieve high availability and avoid system failure, the authors prefer to apply reconfiguration of the system after fault detection rather than increased robustness with performance overheads. In this work the authors do not consider separate modules that are responsible for performance monitoring and error handling.

A System Health Monitoring Unit is used by network-on-chip many-core systems [64]. This unit has an holistic view of the system components' health status. A Mapper/Scheduler Unit generates mapping and scheduling solutions for different classes of faults (transient, intermittent and permanent) that occur in processing elements, on-chip routers and communication links. This approach is bound to the specific network-on-chip architecture and may not be suitable for other architectures such as software applications.

The study [65] aims at providing high availability for the request-oriented distributed system using a *CrossCheck* holistic approach, which extends state-machine replication. This approach employs majority voting based on the hash values of the results, but not on the results themselves to reduce the message size. If some difference is detected by voting, the faulty replica is recovered with a special recovery message. Aspect-Oriented

Programming (AOP) is applied for the protection of critical state-objects to deal with arbitrary state corruption. The experiments proved low performance overhead of this solution. The CrossCheck is intended to optimise performance, but it does not consider the trade-off between reliability and performance.

In this section several approaches to centralised fault tolerance mechanisms were considered. These approaches focus on specific architectures and may not be suitable as approaches for a wide range of systems. In addition, the trade-off between reliability, performance and resource utilisation is not usually considered in such systems.

2.5 Existing approaches to system structuring

This section considers the existing approaches to system structuring, which in some cases diverge from the commonly used methods. All these methods have inherent strengths and weaknesses. In addition, some techniques are efficient or even applicable only in limited application domains. In the following subsections, systems with goal-seeking behaviour, Filter fusion, Integrated Modular Avionics and Ensemble communication architecture approaches are discussed.

2.5.1 Systems with goal-seeking behaviour

The system architecture can be considered from the goalachieving point of view. One example is described by Brooks as the architecture of a layered control system developed for mobile robots [66]. A control system is able to execute many complex processing tasks in real time. Instead of decomposing the problem into functional units, the author decided to apply task achieving behaviour decomposition. Several mobile robot requirements are identified. Firstly, the robot has multiple goals sorted by priority because the goals could be conflicting. Secondly, for navigation purposes, the robot uses multiple sensors, which do not always give very precise data. The third point is robustness. If some part of the control system fails, the robot should rely on working components. Brooks defines his initial motivation stating that it is not necessary to use very complex control systems in order to achieve complex behaviour. *Levels of competence* and *layers of control* are applied to solve each small decomposed subproblem. Levels of competence are defined as a guide for this work. Lower levels implement simple behaviours like avoiding the objects and wander aimlessly without hitting the walls, etc. Each next level offers more

complex behaviour and includes each earlier level of competence as a subset. For each level of competence there is a corresponding layer of control. Layers of control are added incrementally without changing the lower layers. A higher layer augments lower layers of the control system, but the lower layers still produce the results without knowing about the higher layers. The author calls this the *subsumption architecture*. Such an architecture provides additional robustness since the lower levels of competence are well debugged and continue to produce results. If the higher level is unable to produce a result during the specified time, then the lower level will produce an acceptable result. In addition, new layers can be added later if the control system requires additional functionality. The given architecture does not require any central control, because the system is considered as a system of independent agents. However, the lower layers produce results despite the fact that these results will not be used hereafter. Such a scheme lacks system-wide coordination. In some cases, this approach leads to overdesign: redundant operations or waste of resources.

Another relevant work is the idea of the *Teleo-Reactive* programs presented by Nilsson in [67]. To apply this approach, the developer should specify the goal and define the actions to be performed in case of changes in a constantly monitored environment. Monitoring is implemented as continuous computation of the parameters and conditions for the actions. These conditions are in the regression relationship to ensure robust goal-seeking behaviour. The restriction of the Teleo-Reactive programs is that they require a lot of computations to check the conditions. However, the majority of conditions are irrelevant to the current situation or might be predicted very precisely. This approach is not suitable when system resources should be utilised efficiently.

2.5.2 Integrated Modular Avionics

Integrated Modular Avionics (IMA) is quite a new concept intended to reduce aircraft weight and related overheads. It represents real-time computer network airborne systems, consisting of computing modules that support various applications. These applications differ by criticality levels. According to [68], the main goal of IMA is to decrease avionics cost in comparison with earlier design solutions. This is achieved by sharing hardware resources and reducing duplication. Modular architecture of IMA facilitates the development and maintenance of avionics software, since the modules share underlying hardware and use a common API to access this hardware. The concept of IMA is based on powerful

computer processing units – cabinets with a special operating system for independent execution of application software. The hardware is considered as an interface, whereas avionics functions are mostly presented by software modules. The hardware and software are designed independently in order to eliminate their effect on each other. Moreover, software independence is essential to avoid the influence of applications. In IMA systems reliability is achieved by resource redundancy. The IMA systems are mission- and safety-critical which is why these systems must be fault-tolerant and meet rigid task execution deadlines [69].

The paper [70] discusses the transitioning from a federated avionics architecture to an IMA architecture, considering its benefits and difficulties. Resource management in IMA systems is significantly different from that in federated ones. Federated architectures are based on independent collections of dedicated computing resources, whereas IMA architectures leverage avionics functions of various criticalities on a shared computing platform, ensuring weight and power savings and simplifying the development process. The authors note several benefits after transitioning to IMA:

- Flexible allocation of computing resources such as processors, communication network and input/output units. In comparison with federated architectures, IMA allocates resources more efficiently.
- The aircraft weight and power consumption are reduced due to the consolidation of the hardware and associated cooling and wiring.
- Development expenses are reduced, since hardware consolidation means consolidation of development efforts. Developers can concentrate on the application layer, which saves them developing separate architecture for hardware as they do in the case of a federated architecture. “Open” IMA architectures could increase development efficiency by employing interfaces available in the public domain.

The study [70] specifies the factors to be considered before transitioning to IMA:

- Confidence in system-level integration tools and processes. The systems integrator must be able to fulfil critical responsibilities, such as increased interface definition and management, resource allocation and management, system configuration analysis and generation.

- A strategy for legacy system support by defining which avionic functions will be hosted on the IMA platform.
- Establishing an organisational structure that will collaborate to produce an optimised solution for the entire set of hosted avionics functions.
- The approach to resource management will be fundamentally different from a federated architecture. Avionics system developers should work together with the system integrator to attain an integrated solution. The system integrator is responsible for allocating the required resources to ensure performance of each hosted function.

IMA implies fundamental alterations affecting hardware and software layers. In addition, the usage of the IMA architecture assumes a significant organisational shift, putting pressure on teams of developers to work closely with each other. All these activities are not always desirable by development companies.

2.5.3 Filter Fusion

The paper [71] proposes Filter Fusion, a compiler optimisation technique which allows the developer to eliminate the overhead involved in the modular design of independent filters. A filter can be considered as a data-manipulation abstraction that reads the data from the source, processes the data and writes the data to the destination. In a filter application, data flows through several filters, which are separate modular entities. Such a modular implementation has disadvantages in terms of performance. The authors assert that it is more efficient to merge the filters in order to perform all data manipulations at once instead of processing the data in each filter separately. When performance is important, modular design should be replaced by an integrated design.

According to the researchers in [71], network protocol layers are regarded as filters. Programmers usually merge these filters by hand to produce efficient code. If the filters are integrated, the data could be read once, processed many times and stored once, eliminating redundant memory accesses. Network applications require a number of simple manipulations to be executed on each network packet, with the protocol stack formed by these manipulations. The authors claim that it is more efficient to abandon abstractions and merge the protocols for performance purposes.

Filter Fusion integrates complex independently developed filters into a single optimised function. It is more efficient than optimisation “by hand”, since the transformation is

performed automatically and the probability of errors is significantly reduced. Moreover, this approach allows the programmer to retain a modular design without performance deterioration.

The Filter Fusion is designed for the algorithm level and sometimes affects the algorithm of functional components. In this thesis we focus on the approach that would optimise the system without affecting its functional behaviour.

2.5.4 Ensemble communication architecture

Liu et al. [72] consider the pros and cons of systems built from components. They claim that it is possible to significantly reduce end-to-end latency using the Ensemble communication architecture and the Nurpl formal system for program reasoning and transformation.

The authors assert that it is easier to design, develop and maintain a set of individual components with a certain functionality than a monolithic system with the same functionality. Moreover, a system based on components is adaptable to new environments and can be extended in the run-time. However, in the component-based approach, the abstraction barrier between components requires extra overheads. In addition, the installation of the system from components is usually harder than configuring a sealed system.

In this study, the authors use *Ensemble*, a high-performance network protocol architecture based on simple micro-protocol modules, which can be stacked in different ways to satisfy the communication requirements for the applications. These micro-protocols provide flow control, encryption and other network-related functionality. The interface is event-driven: event objects are passed between adjacent modules.

The study [72] explains how Ensemble's micro-protocol components can be used to configure a correct protocol stack. In addition, the authors use the formal tool Nuprl to analyse how the performance of the result system can be improved. The researchers maintain that the following points were crucial for the project:

- It is easier to reason about small and simple components, but not very small.
- Components should be implemented in an event-driven, functional manner.
- A language with formal semantics should be employed.

- Using a formal tool is important, since hand-checking is very difficult and time consuming.

- Collaboration of the system group (Ensemble) and the formal group (Nuprl).

The Ensemble architecture shows how to achieve significant performance improvement using well-designed components and appropriate tools. However, it focuses on the optimisation of system functional characteristics by reducing the overheads of unnecessary data processing and decreasing the end-to-end latency and it does not consider optimisation of FT-related mechanisms. In addition, the Ensemble architecture applicability is limited to communication protocols.

2.6 TCP/IP motivating example

The TCP/IP Protocol Suite [73] acts as a motivating example for HFT. On the one hand, there are benefits in applying a cross-layer error detection and error recovery approach to ensure reliable communication. It illustrates how a well thought-out design assisted in creating an efficient protocol suite that became a foundation of the Internet. On the other hand, TCP/IP still has some architectural flaws, which affect communication performance, especially in wireless networks. These flaws demonstrate the importance of early stages of system design, because the evolution of communication networks was not fully taken into account. The advantages and disadvantages of the TCP/IP protocol suite should be considered when engineering HFT for computer-based systems. This includes the elaborated design of the system at the initial stages of engineering and consideration of future system evolution.

2.6.1 TCP/IP Protocol Suite

The Link layer, the lowest layer of the TCP/IP stack, is used to transmit the packets between the Internet layer interfaces of two nodes inside a local network segment. The Transport layer provides host-to-host communication for the Application layer. The Application layer supports data exchange between processes on different hosts over the network connection supported by the lower layers. Data flow in TCP/IP networks is shown in Figure 2.3.

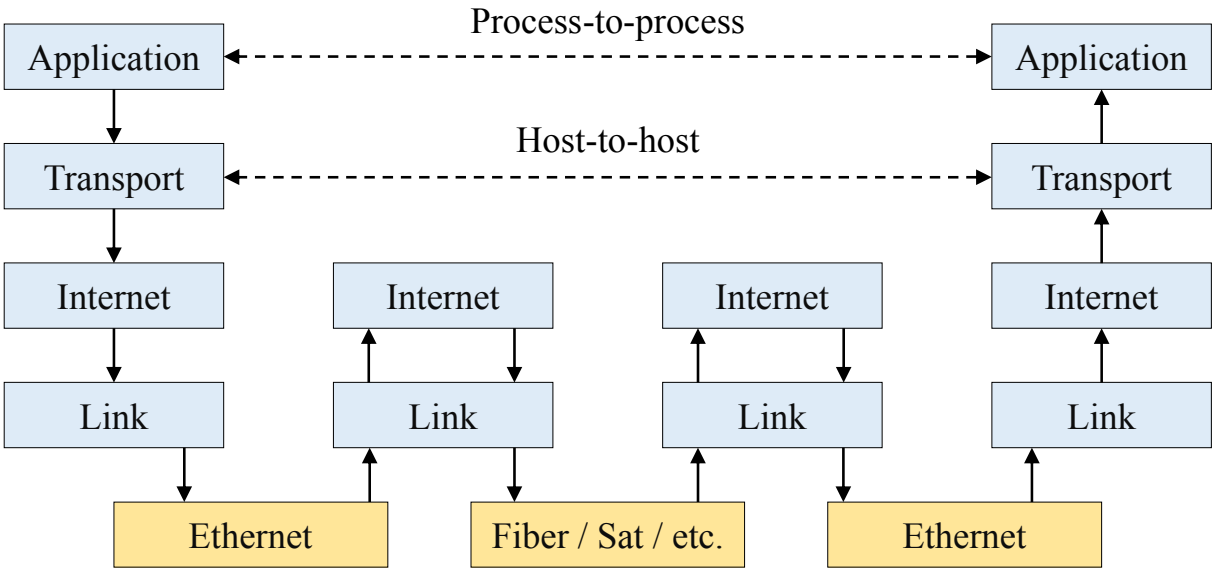

Figure 2.3: Data flow in TCP/IP networks

Transmission Control Protocol (TCP) [74] provides reliable packets transmission, even though the packets may be lost, corrupted or delivered out-of-order. At the Link layer, the Ethernet frame contains a Cyclic Redundancy Check (CRC)-32 checksum. A frame with an incorrect checksum is discarded by the receiver hardware. The main protocol at the Internet layer is Internet Protocol (IP), which has two implementations, IPv4 and IPv6. The header of the IPv4 packet is protected by the CRC-16 checksum. The IP packets with wrong checksums are dropped by the receiver. The IPv6 header does not contain a checksum, assuming that the Link layer provides an adequate error detection. The TCP and UDP packets of the Transport Layer have CRC-16 checksums, which protect the payload and addressing information. TCP sends an *Acknowledgement* to the sender to confirm the correct receipt or a *Negative Acknowledgement* if the packet checksum is incorrect. In the latter case, the Automatic Repeat reQuest (ARQ) method is used to retransmit the corrupted packet. If the sender receives neither an Acknowledgement nor a Negative Acknowledgement by timeout, it resends the packet. Such a situation can happen when the packet is lost or rejected by the lower layers due to an incorrect checksum. In addition, a TCP packet contains a sequence number, which allows the receiver to discard duplicate packets and sequence reordered packets. This, in particular, shows that the errors of the lower layers are detected and recovered by concerted efforts at several layers.

At the Application layer, the developer can choose an appropriate Transport layer protocol: either the connection-oriented and reliable TCP or the connectionless User Datagram Protocol (UDP). The developer's choice between reliable data delivery and fast data de-

livery depends on the application requirements. If UDP was chosen, then it might be necessary to implement error detection and error recovery at the application layer by adding redundant data like status code or custom checksum.

2.6.2 Positive side of TCP/IP

The TCP/IP stack provides an excellent example of CLFT applied to ensure FT and improved performance. All layers of the TCP/IP stack participate in error detection and error recovery in a concerted fashion.

The advantage of the TCP/IP protocol is cross-layer error detection and error recovery that are applied in case of data retransmission issues. The Link, Internet and Transport layers of the TCP/IP protocol use CRC checksum to detect and reject corrupted data packets. In addition, TCP provides data retransmission in the case when the sender did not receive an acknowledgement of successful TCP packet delivery. The following scenario demonstrates cross-layer error detection and error recovery. If the Ethernet frame is corrupted, the data transmission error will be detected by the two layers. The Link layer will detect an incorrect CRC checksum of the Ethernet frame, whereas the TCP sender at the Transport layer will not receive an acknowledgement. Error recovery will be performed by the same two layers. The Link layer will reject the corrupted packet and the Transport layer will retransmit the packet by timeout.

If the developer chooses UDP as a transport protocol then error detection and error recovery are to be implemented at the application layer, since, unlike TCP, UDP does not provide reliability features. In this case, the errors, which are not detected at the lower layers will be detected at the application layer. Depending on the error type, the error recovery could be performed either only by the application layer or in a cross-layer manner by combined efforts of several layers. These examples illustrate how cross-layer fault tolerance is applied to ensure efficient and reliable data transmission over the network.

TCP/IP is a useful example of how FT can be applied in coordination at several system layers. This was done to ensure its efficiency and flexibility. One of the main factors contributing to the success of this protocol is the way the FT was designed and engineered.

2.6.3 Problems with TCP/IP

The TCP/IP protocol is not ideal and has some design issues, which do not affect its popularity, however these issues make TCP/IP less efficient than it could be. There are several major problems that reduce the efficiency of the TCP/IP protocol. These include redundant CRC checks and sending redundant acknowledgement packets at several layers of the TCP/IP stack. In addition, TCP/IP suffers from wrong fault assumptions during the operation in wireless networks making wireless communication less efficient [75–77].

TCP is an abstraction of the network connection to the Application layer. TCP provides end-to-end reliable communication for the application, minimises network congestion and notifies the data source about data transmission failure if the data is still undelivered. Although the TCP protocol verifies the CRC-16 checksum for each segment, the algorithm used is considered weak [78] by modern standards, however this CRC-16 checksum is important because it is an end-to-end checksum. The end-to-end property is crucial, because it means that the checksum is computed by the packet sender and verified by the packet receiver.

Supplementary error detection and error correction techniques may be needed for Link layer implementations with high bit error rates. The use of a CRC or integrity checks at the Link layer partially compensates the weak checksum of the TCP protocol. Nevertheless, the 16-bit TCP checksum is not redundant, since errors in the packets between TCP-protected hops are not unusual [78]. For instance, in 2008, a single bit transmission error spread throughout the system with internal state information message. This error was not detected by network protocols and led to the shutdown and 8-hour outage of the Amazon Web Services servers [79].

The Link layer checksum (CRC-32) is a packet-by-packet checksum and it is recalculated on every network node. Since the Ethernet frame passes through a lot of routers on its way from the sender to the receiver it is dangerous to rely on all these routers assuming that they do not have flaws in the implementation. Thus, both checksums are important even though there could be a possibility to use only one of them. If there was an *end-to-end CRC-32* checksum at the Link layer or the TCP packet was protected by the CRC-32 checksum, an unnecessary checksum redundancy would be eliminated. Actually, when the IPv4 protocol is applied as an Internet layer protocol, there is one more CRC-16 checksum—IP Header Checksum, however it protects only the header of the IP packet.

Finally, as far as CRC-16 is considered weak, additional checks are introduced in some Application layer protocols. For example, a principled hardening technique intended to prevent error propagation and avoid massive system outages due to state corruption is presented [80]. In general, it is strongly encouraged to employ additional tools to guarantee the integrity of the transmitted information for distributed network applications [81].

The Link layer contains two sublayers: the Logical Link Control (LLC) sublayer and Media Access Control (MAC) sublayer. The LLC sublayer has three operational modes: unacknowledged connectionless (LLC1), acknowledged connection-oriented (LLC2) and acknowledged connectionless (LLC3). The LLC2 mode is rarely applied in combination with TCP. However, when this happens several operations are made twice at different layers: flow control, connection establishing and acknowledgement.

Another problem with acknowledgement is that it could be very slow between the remote servers. Three-way handshake is required to establish a connection. If the latency between two network nodes is high then connection establishment could take a sufficient amount of time.

Head-of-line blocking is a problem that occurs when a stream of packets is blocked by the first packet in the stream. This could happen if the first packet in the sequence is lost or corrupted. Even though other packets are already delivered they cannot be processed because TCP is obliged to deliver data in the order it was sent.

TCP uses an end-to-end flow control protocol to ensure the sender is not sending data faster than the TCP receiver is able to process the data. This is necessary because network nodes operate at different speeds. Flow control at the Transport layer is implemented according with sliding window flow control protocol. The receiver of a data packet specifies the maximum amount of data (receive window) in the acknowledgement packet that it is able to receive. The sender is allowed to send only up to this amount of data. Then the sender waits for an acknowledgement with a new window size. If the window size is 0, then the sender stops data transfer to allow the receiver buffer to process the income traffic.

Another feature of the TCP protocol is congestion control preventing network congestion collapse due to network performance degradation [82]. Congestion control mechanisms that manage the data flow in the network include the following algorithms: slow-start, congestion avoidance, fast retransmit and fast recovery. The slow start algorithm is a non-efficient solution for small objects, because it prevents using available network throughput.

Flow control and congestion control are usually efficient in wired networks. However, in wireless networks the situation is different. Bit-error rate in wireless networks is much higher that in wired networks. This happens because wireless networks are prone to occasional packet losses due to various factors like signal loss, interference or user mobility. Usually these reasons are not related to network congestion. The TCP protocol was initially designed for wired networks, and the packet loss is considered due to network congestion rather than wireless signal problems. After the packet loss the congestion window size is significantly reduced to prevent congestion. The throughput of the connection is reduced as well. If there was any evidence that the packet was lost because of wireless problems rather than network congestion, the wireless performance would be much better. Several solutions [83, 84] have been offered to overcome these issues in wireless networks.

2.6.4 Lessons learned from TCP/IP motivating example

The TCP/IP example demonstrates the advantages of an holistic view and applying a cross-layer approach. At the same time, this example points out how early design flaws, lack of system-wide analysis and wrong assumptions lead to unnecessary overheads and finally affect performance and quality of service.

Advantages and disadvantages of the TCP/IP protocol mentioned in this section act as a very good guideline to be followed during engineering of fault tolerant systems. The design of such systems should provide an easy way of applying diverse FT mechanisms. At the early stages of system engineering developers should consider how error detection, error handling and fault handling can be designed and implemented in an holistic fashion using all the benefits of cross-layer and centralised approaches to system-wide FT. Fault assumptions, error propagation chains and the system-wide implications of errors must be considered at each stage of the system life cycle. Possible future errors as well as expected system evolution should always influence system requirements, design and implementation. After thorough analysis and elaboration of system design, redundant interactions and computations that do not contribute to system dependability and efficiency should be removed whenever possible.

To conclude, TCP/IP is the basis for communication in the Internet and Intranet networks. It has many advantages that make it useful and flexible. Nevertheless, even such an elaborated and successful protocol is vulnerable to architectural flaws and imperfections that were made during the design time and now they are very expensive to fix because of

its prevalence.

2.7 Conclusion

Computer-based systems are prone to faults at different layers of the system stack, starting from circuitry degradation at the hardware layer to the bugs in the application source code. In designing large systems substantial efforts are being made to mitigate the effects of errors caused by faults at all layers of the system stack. Traditionally, the errors are mainly handled by those components, which detected these errors. On the one hand, this approach is convenient, since error recovery is attempted "on the spot" and—in case of successful recovery—the error will not affect the remaining part of the system. On the other hand, the system-wide implications of the error are not usually analysed, and the necessity of handling is not typically considered. This situation illustrates the predominance of convenience over system efficiency in the run-time.

Thereby, when the system efficiency becomes crucial, the system architect should consider a transfer from conventional ways of system structuring to achieve efficient resource utilisation and retain system reliability. The efficiency of the system FT could be achieved only when all crucial system units, such as modules, components, or layers are considered together during the engineering of system FT. However, it is very difficult to achieve when the system units are engineered separately without an holistic view of the system.

In this chapter the fundamental theories on dependability and fault-tolerant computing were analysed. In many situations these theories are successfully applied to develop dependable computer-based systems. However, as shown in Chapter 1 there are still many examples when developers do not follow the rules and guidelines due to the system complexity, underestimating of FT mechanisms, and strict deadlines in the fast-paced development environment pushing the developers to focus on functional requirements, while non-functional requirements are neglected. This reduces understanding and maintainability of FT mechanisms. In addition, very often the developers focus on separate components when implementing system FT mechanisms without considering the entire system. This is the reason why the system-wide view on the trade-off between reliability, performance and resource usage is not usually analysed making the systems less efficient than they could be. The several approaches to FT engineering and system structuring that are intended to facilitate maintainability and efficiency of system FT mechanisms

were also examined. All these approaches have certain limitations, they are either applied in restricted application domains, suitable for specific architectures only or do not focus on efficiency or maintainability of FT techniques. The TCP/IP motivating example has shown the advantages of an holistic view of the system and importance of expected system evolution analysis to ensure that the system will be efficient in the future.

The existing research, that tackles the problems stated in Chapter 1, along with a critical analysis of the existing research evidenced that the problems of FT efficiency and FT maintainability are not fully addressed. Therefore, there is a need for a systematised engineering approach that addresses the efficiency and maintainability problems related to the FT functionality. This approach should encompass the benefits of the cross-layer and centralised approaches to FT, and overcome their limitations, while providing an holistic view to the system FT during all stages of the system development life cycle including predictions of the system evolution. This approach based on the concept of Holistic Fault Tolerance will be considered in the next chapter, where the engineering steps that should be undertaken in order to design and implement a system based on the HFT architecture are described.

3

Engineering Holistic Fault Tolerance

Contents

3.1 Introduction

In Chapter 1 the importance of system FT, the cost of design flaws in the system FT, complexity of FT and examples of FT misuse were discussed. In Chapter 2 the approaches intended to tackle these issues were considered. However, to the best of our knowledge there are no suitable approaches that focus on the efficiency and maintainability of FT mechanisms simultaneously.

In the previous chapter the background and motivation for this research project were provided. All conventional ways of ensuring system FT are associated with a component structuring. The most prominent forms are nested atomic transactions [53], nested recovery blocks [50] and nested exception handling. Techniques like these help developers to focus on providing FT only at one layer/component, either by tolerating faults (errors) or, when this is not possible, by passing the responsibility to a higher layer. This view is best captured by the IFTC architectural style [55]. These recursive approaches are widely used because they allow developers to focus on providing FT of system components. However, it seems that these solutions do not always support a system-wide view on ensuring FT, sometimes causing system overdesign and resulting in systems that are not efficient with respect to system performance or resource usage. These approaches are not usually applied to sacrifice reliability for performance or vice-versa depending on the system operation. Thus, the trade-off between reliability, performance and resource utilisation is not usually considered.

There are several other structuring approaches that are different from the conventional architectures but they cannot fully support a system-wide coordination of the FT mechanisms. Brooks' approach [66] to structuring complex control systems using levels of competence is an interesting solution but it supports neither component (in this case, level) coordination nor activities that require dynamic decisions which groups of components to involve in FT. The IMA architecture is not developed for modular coordination either. These approaches are not intended for dealing with situations when several system components need to be involved in system-wide FT in an orchestrated fashion.

There is an interesting evidence indicating that the TCP/IP design that motivates our work (Section 2.6) suffers from certain weaknesses because arguably there were some mistakes made in designing CLFT [75]. In certain cases, error recovery can be inefficient due to a lack of collaboration between the Link and Transport layers, as the TCP considers

timeouts to be a consequence of high network load and slows data retransmission accordingly. However, in wireless networks the data packet can be lost due to environmental conditions and user mobility, so the lack of information about an error can cause data transmission delays. This means that, for system-wide FT to be efficient, practical and reusable, its engineering needs to be properly supported.

These issues can be addressed by implementing a vision of HFT, that introduces a new type of cross-system management to capture system-wide FT. The local FT of system components would be complemented by a system-wide FT and fault handling strategies. HFT would not only simplify the analysis and implementation of critical system parts, but also facilitate system reuse and elaboration of system operating modes.

The concept of HFT is introduced in this chapter to deal with problems formulated in Chapter 1 and overcome the limitations of existing approaches considered in Chapter 2 that tackle these problems. The concept of HFT does not involve the introduction of new or the change of the existing well established techniques [55]. The idea is to support reasoning about the system FT at the system level rather than at the level of individual units and apply error detection and recovery techniques at the system level to make FT-related mechanisms more maintainable, modular and reusable. It should be noted here that FT and maintainability concepts are closely interconnected. FT is a means of dependability, whereas maintainability is an attribute of dependability [1]. The concept of HFT focuses on both of them to ensure dependable and efficient operation of the system.

This chapter will consider the systems engineering and FT engineering principles in Section 3.2. The concept of HFT will be introduced in Section 3.3. The principal steps that are required to engineer HFT for a computer-based system will be described in Section 3.4. Then, the benefits of HFT and the challenges to be faced by the developers during the engineering of HFT will be outlined in Section 3.5. Concluding remarks of the chapter are given in Section 3.6.

3.2 Systems engineering

Engineering defines how science, empirical evidence, and mathematical methods can be applied to design, development and maintenance of different systems, devices, objects, or processes. The engineering discipline covers a great variety of engineering fields.

Systems engineering is a field of engineering for creation and maintenance of complex and

successful systems throughout the system development life cycle [85]. Systems engineering considers various issues throughout the system life cycle, such as requirements engineering, system dependability, cooperation of developers, verification and evaluation. All these tasks are very difficult for the engineering of large and complex systems.

In this section, software engineering and FT engineering principles will be considered.

3.2.1 Software engineering

According to the International Standard [86], software engineering is defined as *"the systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software."*

Various paradigms and development models can be applied to engineer software systems from the conservative *Waterfall* model to the modern *Agile* model [87]. These models differ in development practices, flexibility of the development process, customer involvement, approach to testing, philosophy, and system evolution vision. However, all these models include typical core activities, such as requirements elicitation, design, implementation, testing, verification, exploitation and maintenance.

FT is an essential part of all engineering steps. The system developers should pay significant attention to engineering system FT. The FT mechanisms for the system are chosen according to the system requirements. Fault assumptions, abnormal behaviour in component specification, error detection, system structuring for FT should be defined at the earliest stages of the software engineering and clarified throughout the development life cycle.

3.2.2 Dependability and fault tolerance engineering

FT is complex to implement and maintain and also costly due to redundancy requirements, however the cost of failure should also be considered. A systematic use of FT mechanisms during development of computer-based systems is a part of an engineering process. Dependability and FT are important quality factors of computer-based systems.

Dependability and FT engineering usually are going along with system engineering. Moreover, system dependability and FT should be ensured at every stage of the development life cycle.

The methods for systematic engineering of dependable and fault tolerant systems are considered in this subsection. The main engineering steps and activities in each of these steps are considered in detail.

Avižienis [88] discusses the issues arising during the FT provision and ensuring its effectiveness on the example of dependability requirements of an Advanced Automation System. This system must meet extreme availability and safety requirements with less than 3 minutes down time per year. Such systems require a systematic and hierarchically structured introduction of FT techniques during the entire design process. The rigorous design and evaluation guidelines are intended to facilitate this process. During the system design FT should be considered as a fundamental issue, which includes tolerating human-made design and interaction faults, the balancing of performance and FT, and the integration of subcomponent recovery activities into a multi-level recovery hierarchy. The introduction of FT into the system should be accompanied by a methodical approach starting with initial design concepts, which requires collaboration of performance and fault tolerance architects during the definition of critical tasks and subcomponent communications. Such an approach is called the *design paradigm* for fault-tolerant systems. The main steps of the paradigm include:

- definition of the expected fault classes over the system lifetime.
- specification of performance and dependability goals, which could vary for different operation modes.
- partition of the system into subsystems for performance and FT implementation.
- choosing error detection and fault diagnosis algorithms for every subsystem.
- elaboration of recovery algorithms for every subsystem.
- evaluation of FT effectiveness and its impact on performance.
- integration of subsystem FT on the system-wide scale.

These activities significantly assist in meeting the stated dependability goals. The design paradigm is aimed to reduce the likelihood of oversights, mistakes, and inconsistencies during the process of meeting the specified dependability goals by using selected implementation means of FT. A successful design of a fault-tolerant system requires a sound

verification of the design completeness and meeting the specified availability and safety goals. During the verification it is necessary to ensure that there is no unnoticed unjustified simplifications, such as insufficient fault assumptions or implicit assumptions that all FT functions are not faulty themselves. In addition, it should be checked that the effectiveness and speed of error detection, fault diagnosis, system state recovery and reconfiguration are not overestimated. The verification should contain qualitative and quantitative evaluation. The former is responsible for fault assumptions validation, FT attributes completeness verification and checking of proper integration of subsystems recovery procedures. The latter includes determination of the ranges of coverage and execution time parameters for all FT functions in all subsystems. Even with such a complete verification there are the challenges to consider: tolerating new fault modes, dealing with unanticipated failure inducing interactions, handling more than one concurrent fault activation, tolerating residual design faults, extension and reverification of the system protection after every system modification or repair. The summary on how HFT addresses these challenges will be given in Section 8.2.

It is pointed out in [17] that there is a need to engineer FT in a disciplined and rigorous way. Starting from the requirements phase, different faults and appropriate FT mechanisms to deal with these faults should be considered and refined during the system life cycle. The FT verification and validation should become a part of system development. The FT mechanisms under development should correspond to the types of faults, application domain, execution environment, system requirements and development model to avoid unnecessary complexity of the system. In addition, the FT mechanisms should be ready for system evolution to ensure that all modifications of the system, including the FT mechanisms, would be smooth and dependable.

A framework for dependability engineering of critical computing systems is presented in [5] where two major classes of processes are considered. The first is the *system creation process*, which includes classical engineering steps. The second is the *dependability processes* that are responsible for dependability means creation. These processes are used to support the development of dependable systems.

It is established that design of dependable systems delivering critical services is a complex task because of computer-based systems' complexity and the diversity of fault classes. This requires a systematic and structured design approach integrating dependability concerns starting from the earliest stages of the system engineering process. It is noticed in [5]

that traditional software development models focus on verification and validation, and do not incorporate reliability evaluation or FT. The authors propose to explicitly incorporate the dependability means in a development model to produce dependable systems. The complementary activities supporting the dependability means can be appointed to separate processes working in line with the system creation process, that orchestrates all the activities. The system creation process is based on traditional engineering steps including requirements elicitation, design, implementation and integration. Requirements elicitation includes the specification of the systems representing the user needs. The system architecture is defined during the design step and system components are implemented according to their specification at the implementation step. The integration step is responsible for the system integration into its operational environment.

The dependability processes comprise fault prevention, fault tolerance, fault removal and fault forecasting processes. Each of these processes corresponding to dependability means consists of several activities. For example, the fault prevention process includes evaluation of risks from the system development and project breakdown into tasks, whereas the FT process consists of system partitioning and fault and error handling activities. The fault removal process is composed of system verification, system diagnosis, and system modification activities. The fault forecasting process includes the statement of dependability objectives, allocation of these objectives among system components and evaluation of the system behaviour.

The authors [5] state that the development of dependable systems requires the interactions between the system creation process and all the dependability processes including successive analysis and iterative refinements throughout the system development life cycle. For instance, the interactions between the fault forecasting and the FT processes define the acceptable operation modes and constraints for each mode.

Fault assumptions are crucial in dependable systems development [5]. It should be noted that fault assumptions related to different dependability processes are usually not identical. Some faults are tolerated, while others are forecasted or removed.

The authors [5] propose the following systematic approach to engineer fault tolerant computer systems. A detailed description of the main system functions including dependability goals should be defined at the requirements elicitation step. In addition, different development and certification constraints are defined during this step. The goal of the FT process during the requirements step includes the identification of undesirable events,

description of the system behaviour in case of failure and definitions of mechanisms to satisfy the dependability goals. It is necessary to consider how to deal with more than one concurrent failure. Since the system quality attributes like reliability, availability and performance influence each other, it is impossible to optimise all of them simultaneously. That is why appropriate trade-offs between these properties and FT mechanisms should be considered during the requirements step in order to facilitate the implementation of FT mechanisms. The fault forecasting process during the requirements step assesses the occurrence of failures and classifies the system functions by criticality levels. Based on this information, appropriate error handling and fault handling mechanisms will be chosen during the design step. These data may be obtained from similar systems or from the system under development if such information is already available.

The main goal of the design step [5] is to specify an architecture that allows the system to meet the elicited requirements. During the design step, the system structure consisting of separate components, the system behaviour, and data flows between these components are defined. The system component can be considered as a system as well. That is why the decomposition is recursive and it continues until the low level components are considered atomic. In spite of the fact that there is not a universal approach to system decomposition, the design should consider future system evolution, testability, maintainability and FT.

In an ideal case, the design should consider all classes of faults with corresponding FT and fault avoidance mechanisms. A special attention should be given to formal specification and verification of FT mechanisms, non-fault tolerant components and single points of failure. Error confinement areas and fault independent areas should be defined during the FT structuring. The former assists in the definition of error propagation barriers between components, whereas the latter identifies components whose faults are not interconnected. Apart from error handling mechanisms definition, the design should consider fault processing mechanisms, which include fault diagnosis and system reconfiguration. Depending on the design choice, error handling and fault handling mechanisms can be distributed across the system or centralised. The FT mechanisms are chosen based on the type of the system components, fault assumptions, and dependability attributes to be satisfied. In addition, special techniques are required to protect FT mechanisms themselves. It is preferable to use verification techniques for the design of components that represent a single point of failure. In parallel with the design of the system architecture the techniques for system integration testing should be defined.

The fault forecasting process during the design step should include dependability evaluation in order to check that the chosen architecture meets the specified dependability goals. Analytical modelling and simulations can be used as evaluation measures. The risk of introducing too many FT mechanisms should be considered during the design step. To evaluate the efficiency and sufficiency of FT mechanisms a fault injection technique can be applied.

The design and implementation steps are significantly affected by the expected system evolution. Future modifications of the system are much easier and less expensive if the requirements evolution is anticipated. However, it demands an active involvement of the system end-users.

This section considered the guidelines for dependable systems engineering. The similar guidelines and advice should be considered during the engineering of HFT.

3.3 The concept of HFT

This section introduces the concept of Holistic Fault Tolerance, states the goals of HFT and describes its main characteristics. The presented concept is called *holistic*, because it is proposed to consider an entire system during the design of the system's FT. FT was chosen as a main part of the concept, since it is an important crosscutting concern, which can affect other non-functional properties of the system. HFT should allow the system to operate efficiently with regards to performance and resource usage while ensuring dependable operation and convenient maintainability of crosscutting functionality.

The concept of HFT pursues two goals. First, it would allow developers to design and maintain complex and reliable computer-based systems in a more coherent fashion than the conventional structuring techniques by supporting a disciplined and systematic way of capturing and modularising the crosscutting functionalities related to error detection, error recovery, and fault handling. The second goal is to achieve an efficient system operation based on reasoning about the interplay between reliability, performance and resource utilisation at the system level rather than at the level of individual system components or any other structuring units used (such as layers, classes, etc.).

The concept of HFT assumes that the system consists of components providing the main system functionality. The *critical components* are those whose operation affect the reliability and performance of the entire system. The non-critical components are responsible

for auxiliary functionality and do not affect the main service provided by the system. For example, complementary user interface features, information loggers, components performing optional, postponable tasks. Even though these features improve the usability of the system, critical functionality can be provided without them. A broader discussion on critical components, different criticality levels, and separation of critical and non-critical tasks can be found in [89]. Thus, the FT mechanisms that do not affect the system-wide dependability and efficiency are implemented and managed by the system components themselves. However, the FT mechanisms that affect the entire system with regards to dependability, performance and resource usage should be managed by the modules responsible for HFT. HFT should not significantly affect the functional part of the system since its goal is to focus on ensuring dependable and efficient operation of components comprising the system.

According to the vision of HFT, the mechanisms implementing non-functional requirements of computer-based systems should be designed and developed taking into account the entire system. Various decisions regarding system reliability, quality of service, performance or resource usage should be made after system-wide analysis. It is not practical to focus only on reliability, performance and resource usage of individual components, since the picture could change when these components form a system.

The idea of HFT implies that the developers of computer-based systems should use dedicated modules to capture the essence of HFT. That includes system-wide monitoring of non-functional properties, analysis of the trade-offs between these properties, and implementation of global crosscutting FT mechanisms involving critical system components. They will create such systems together with dedicated HFT modules that will define FT policies which crosscut system functionalities. This would allow the developers to deal with complexity of the FT mechanisms, facilitate maintainability of the FT mechanisms and ensure efficient operation of the system.

To be flexible, the concept of HFT should support operation modes and load balancing capabilities. These features will be considered in the following subsections.

3.3.1 Operation modes

Operation mode is a special state of the system. Operation mode defines which system functionality will be available at the given instant of time. Or in other words, operation mode determines a link between the system state and available system functionality, since

the capabilities that are available in one mode may not be available in another one. For example, a different set of operations is available for an aircraft during the taxiing, take-off or flight.

The concept of HFT advocates that associating modes with various levels of performance and resource usage is a useful and practical way of structuring HFT in computer-based systems. As has been shown [90], FT can be linked with the system modes in a natural way.

In [91] the authors promote the notion of mode as partition of the system state space. In addition, they consider the modes as a convenient method for modular specification of large state machines. Transition is used to change the mode for the system. Two possible relationships between modes are provided: serial and parallel. Serial mode means that the system could only be in one mode at one point of time, whereas parallel relationship assumes that the system is in all of the available parallel modes simultaneously. Thus, modes represent control information on system behaviour.

Modal systems are presented in [92]. In this study a modal system is defined as an abstract specification of the modes and mode transition. The authors propose formal definitions of the abstractions for specifying modal systems. It is claimed that operation modes are very common in real-time systems, for example a deadline could depend on the operation mode. Using modes and modal system refinement facilitates the definition of system properties, transformation of system requirements into model definition, and control structure of the system.

To this end, engineering mechanisms for mode design and analysis that are associated with HFT will need to be developed.

3.3.2 Load balancing

Load balancing [93] is responsible for workload distribution across several computational resources such as clusters, networks, CPUs, or data storages. The goal of load balancing is to optimise resource usage and increase system throughput. The system reliability and availability may be improved through redundancy if the system has several components performing the same tasks and implements load balancing mechanisms. However, these mechanisms should be implemented carefully, since they should be reliable as well, and depending on the system they may require dedicated software and/or hardware. It is proposed to provide load balancing capabilities by HFT.

3.4 The HFT engineering stages

In the previous section the concept of HFT was presented. This section considers the steps required to engineer the system with HFT based on the guidance presented in Subsection 3.2.2.

The main steps of HFT engineering are illustrated in Figure 3.1 (the key HFT-related steps are explored in detail in the corresponding chapters). After the requirements elicitation phase of the engineering process, the HFT architecture is designed. Then, at the modelling step, the HFT elements will be elaborated and tuned efficiently and all important interfaces between them and the system components will be defined. Finally, a designed system will be ready for the implementation step. Successful implementation is followed by verification and testing.

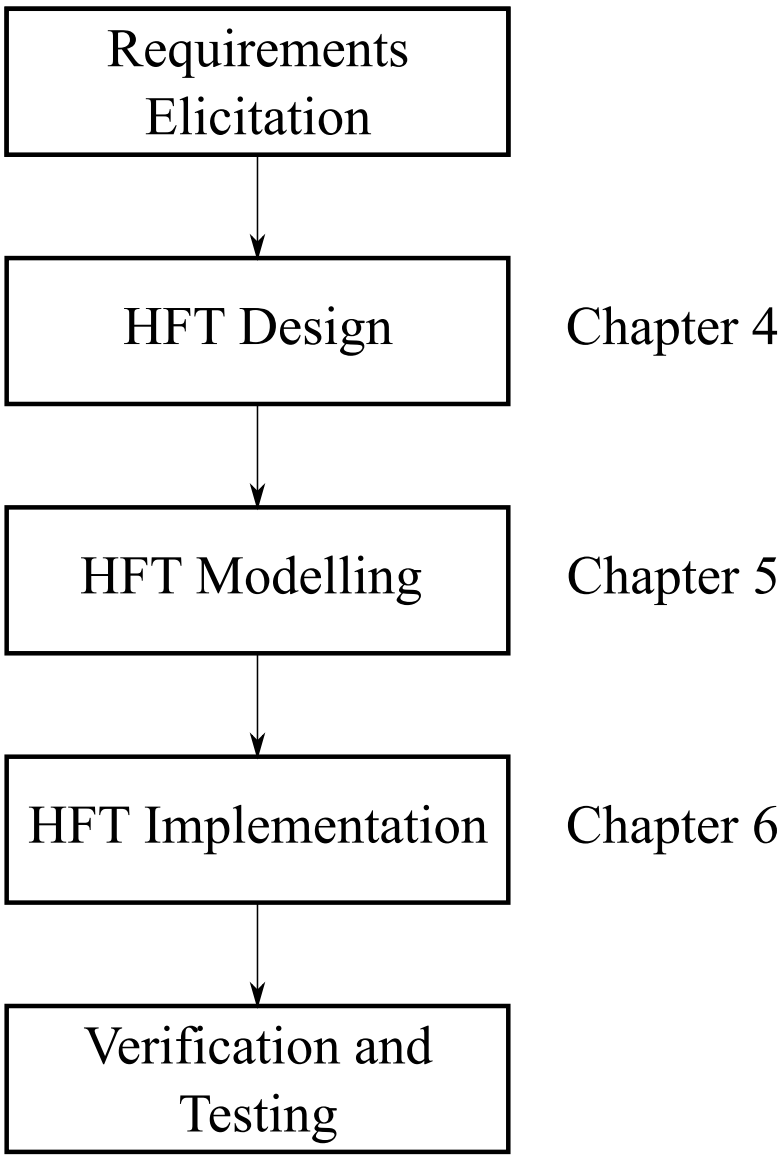

Figure 3.1: The HFT engineering stages

3.4.1 Requirements elicitation

HFT engineering starts with requirements elicitation and preliminary analysis of the proposed system. At this step, the problem to be solved is stated and functional specification of the system is produced. In addition, the operation environment and development constraints of the system are defined. The system dependability attributes and their trade-offs are stated according to the system non-functional requirements. During the requirement elicitation step system fault assumptions should be defined and possible FT mechanisms should be proposed. After that, operation modes and reconfiguration policies of the system can be defined.

3.4.2 The HFT design

Apart from satisfying functional requirements, the design stage of HFT engineering should ensure efficiency and maintainability of the future system. This stage defines the system architecture including the HFT elements. Description of the system behaviour and decomposition of the system to components are made. Depending on functional requirements, critical system components are defined and designed.

During the design stage it is necessary to address several issues with regards to system FT. Fault assumptions defined during the requirements stage should influence the HFT design decisions including the structuring of the component-wide and system-wide FT mechanisms. In addition, error handling and fault handling techniques should be chosen. After that the interactions, including interfaces and data flows between the HFT part and critical components, should be described depending on the elicited system requirements.

The designed system should be analysed to determine the possibility of managing performance and resource utilisation by introducing special knobs and monitors. These mechanisms will be controlled by the HFT part of the system.

3.4.3 The HFT modelling

The HFT modelling step should assist in the refinement of the HFT part and its interfaces with critical system components. To do this, critical system components should be implemented and characterised to analyse their performance, reliability and resource usage. This would allow the designer to select only the necessary interfaces between the HFT

elements and the system and to refine the structure of the HFT elements. This should be done before the implementation of HFT for the system.

3.4.4 The HFT implementation and verification

When the HFT architecture is designed and refined, the system implementation can be started. During the implementation stage, appropriate techniques and tools are chosen to satisfy the requirements and develop the system according to the designed architecture. The HFT implementation is followed by verification and testing. However, the last steps do not require specific HFT-related techniques. In this way standard verification and testing techniques will be applied.

3.4.5 The HFT engineering support

Various approaches can be used to support the HFT crosscutting design, including AOP, dedicated reference architectures, extended programming languages, programming libraries, patterns and styles.

Model-based development will play a crucial role in applying HFT for computer-based systems. This will allow analysis/verification at the earlier architectural phases and support direct traceability of designs to critical system requirements, such as safety, reliability, efficient resource usage and system performance, which can only be met by applying FT. Note that other non-functional requirements, such as confidentiality and execution cost, are related to FT and the use of redundancy, too, so they can also be met by designing HFT. Modelling HFT can be supported by the mechanisms available for enabling architectural viewpoints (e.g. [90, 94]), because HFT is in effect an architectural view on system-wide FT.

It is clearly not practical to develop HFT that would deal with all possible errors in the system. Its complexity would be overwhelming. The practical question is to decide which errors to handle locally in components and which to handle holistically. The mechanisms to be developed should support both types of FT and their smooth integration and interplay.

Modelling techniques for HFT should support zooming into system models to help with exploring various HFT solutions and choosing which system components to involve in HFT (again, it is not practical to link HFT with all system components). One promis-

ing technique to ensure HFT scalability is Order Graphs [95], which support a rigorous selection of the level of detail at which models are reasoned about. In terms of resource usage, the efficient way is to focus only on components that require a substantial amount of computational resources [96].

The HFT architecture should describe a set of system-wide FT policies along the lines of a well-established area [55]. This will require a notation that can represent types of faults and errors and their combinations, the corresponding error recovery and fault handling techniques, resources required, sets of system components involved and error propagation chains. For complex systems this could even be done with assistance of special Domain-Specific Languages (DSL). The application of such DSLs will help HFT developers to make their work less error prone and more reusable.

To explicitly control performance and resource utilisation, HFT should be able to use a number of knobs in different system components. This includes switching off/on CPU cores, excluding CPU cores at the OS level (e.g. using Linux governors), using Dynamic Voltage and Frequency Scaling, etc. In addition, various application-specific solutions can be employed: using less precise computation or reducing the quality of the outputs. These need to be supported as well.

Engineering HFT is a complex problem and it is a part of the system development life cycle. Thus, system developers need support, recommendations, and guidance how to engineer HFT throughout the system engineering process. The main challenges of HFT engineering to be encountered by the developers are considered in the next section.

3.5 Challenges of the HFT engineering

HFT always relies on a centralised design in which the HFT control is mainly defined in the HFT controller (Section 4.3), but there might be various ways of implementing the HFT controller. A centralised component is the first option but this can create a single point of failure if the component crashes. This can be solved by standard FT techniques like component replication and checkpointing. Another option is to attach an HFT controller to every critical component during the implementation step, so that each component is controlled by its own controller.

As part of the HFT design it is essential to generate a table that would compile all the faults that have to be dealt with at the system level, and the corresponding error

detection, error handling and fault handling strategies, which can involve several system components. It is important to include in this table the erroneous situations when two or more faults can activate in parallel.

The system-wide FT mechanisms require a special design of the system to ensure that all necessary properties of the system are captured by the HFT part. After definition of the system components and the HFT elements there is a need to refine the architecture to ensure that all the interfaces between the system and the HFT part are necessary and they contribute to system reliability and system efficiency. To do this HFT modelling should be applied.

The HFT mechanisms require special implementation that addresses system crosscutting concerns, while providing good understanding of these mechanisms. They should be easily maintainable in spite of the fact that they are managed by the centralised HFT control. The guidelines on how to implement the system with HFT will be given in Chapter 6.

This section identified the problems that could arise during the engineering of HFT. To deal with these problems a global workflow for HFT engineering was presented. Some engineering steps, like requirements elicitation, verification or testing do not require a special attention for HFT, however, such steps as architecting (Chapter 4), modelling (Chapter 5) and implementation (Chapter 6) should be supported. The following chapters will consider the major steps of the HFT engineering workflow presented in Section 3.4 and provide answers to the challenges discussed in this section.

3.6 Conclusion

Since FT engineering is a complex task it requires a systematic and structured approach during all stages of the development life cycle. The FT mechanisms should be taken into account during all these stages.

This chapter introduced a concept of HFT, considered important stages of engineering a dependable system and provided an outline of the steps required to design and implement a computer-based system with HFT.

Contributions of the proposed HFT include a centralised access to crosscutting functionality, which should result in better maintainability of FT mechanisms, support of operation modes and load balancing capabilities, and prerequisites for efficient system operation.

The chapters following correspond to the three major steps of HFT engineering and will consider how to architect, model and implement HFT for a computer-based system.

4

Holistic Fault Tolerance Architecture

Contents

4.1 Introduction

In the previous chapter the concept of HFT was proposed. To build a computer-based system in accordance with this concept there is a need to support HFT engineering throughout the system development life cycle. The design step is crucial since the fundamental architectural choices are made during the design process. This chapter proposes an architectural pattern for systems with HFT. It introduces the HFT architecture, describes all its elements, interactions between these elements, and considers how to apply the proposed architectural pattern for computer-based systems.

To apply the HFT architecture it is assumed that the system is built out of components whose responsibility is to deliver the main system functionality. The core of this architecture is a special unit called the *HFT controller*, which is supported by a number of *HFT agents*. These together ensure dependable and efficient system operation. In addition, they facilitate the management and maintainability of the system FT mechanisms. The HFT controller coordinates system-wide FT and performs required system reconfigurations, while the HFT agents assist the HFT controller by simplifying its implementation and improving its scalability. Each HFT agent acts as an intermediary between the HFT controller and one or more system components. This was done to reduce the complexity of the HFT controller.

This chapter is organised as follows. Section 4.2 provides the background on software architecture, architectural styles and patterns. Section 4.3 and Section 4.4 describe the HFT controller and HFT agent correspondingly. Cooperation between the HFT controller and the HFT agents is considered in Section 4.5. Section 4.6 explains the interactions between the HFT elements and critical system components. Discussion and recommendations on the design of computer-based systems with the HFT architecture are given in Section 4.7. Finally, Section 4.8 concludes the chapter.

4.2 Background

An architecture is an unchanging deep structure. The term *software architecture* was motivated by the architecture of a building [97]. The main concepts of a software architecture include decomposition of the system into components, describing interactions among these components, and satisfying the system requirements (functional and non-functional).

4.2.1 Software architecture

There is a great variety of software architecture definitions. Bass et al. [98] define a software architecture as *"the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them."* It focuses on external properties of system elements, whereas private implementation details of these elements are not architectural. The software architecture considers services, which are provided and required by the component, system performance characteristics, system resource usage characteristics, the FT and fault handling mechanisms.

The software architecture defines system components and interactions among them [99]. To specify the system structure and topology the architecture illustrates the compliance between the system requirements and components of the designed system.

Three constituents of software architecture are defined in [97]. The first is *elements*, that are presented by processing elements, data elements and connecting elements. The second is an *architectural form* describing properties and relationships between these elements. The third is a *rationale* motivating the choice of architectural style, elements and the form.

The software architecture is very important since every quality system requires a solid foundation. It represents a high level structure of a software system [100] and focuses on early high-impact design decisions, that are very difficult to change. Thus, the right architecture is crucial to develop a successful system. Abstract representation of the architecture improves system understanding, simplifies analysis of complex systems, and facilitates negotiation and communication among stakeholders and engineers. The software architecture addresses crucial properties of the system such as performance, maintainability and dependability. When done correctly it can be successfully reused for future systems.

4.2.2 Architectural styles and patterns

An *architectural style* [100, 101] is a collection of architectural design decisions for a particular development context. An architectural style defines structural organisation of the system, including components, connectors between these components, and constraints imposed on components and their connectors. Some examples of architectural styles include component-based system, client-server and pipe-and-filter.

An *architectural pattern* [101] is a reusable solution for general problems in software architectures. It defines architectural design decisions for typical design problems. For example, MVP and MVC patterns are applied to separate data and presentation.

Some researchers and practitioners do not distinguish between architectural styles and patterns. For instance, Garlan and Shaw [100], and Microsoft guide [30] treat architectural styles and architectural patterns as synonyms. However, these concepts can be distinguished by their scope. An architectural style is a conceptual way to organise the source code, whereas an architectural pattern provides a solution for recurring problems.

In addition there are *software design patterns* [102] which are specific to particular parts of the source code. For example, the famous patterns are: *Factory* (creates class instances when a class type would be known at run-time only), *Façade* (provides a unified interface to a big part of code making it easier to use), *Singleton* (guarantees that there is only one instance of a class), etc.

In case of HFT, it is an architectural pattern that solves the problem of complexity, maintainability and inefficiency of FT mechanisms. It can be applied in combination with different architectural styles and patterns considered in this subsection.

4.2.3 Reference architectures

A reference architecture is a software architecture that provides a typical architectural solution in a particular application domain [103]. It is a template based on the generalisation of the architectural structures that were applied in successful implementations and can be referred to for best practice. It is a guidance for solving the same problems of development, standardisation and evolution of a system based on past experience. It could speed up the system engineering due to reuse of existing architectural solutions and improve system interoperability because of using a proven standard solution. In addition, knowledge based on past experience allows the development team to avoid typical design flaws. A reference architecture encompasses hardware, software, development process, system requirements, system configurations and system components. It should evolve and include new experience and practices to be up-to-date.

A proposed HFT architecture could be used as a reference architecture in particular domains after several successful implementations of systems with HFT in these domains. It will be considered as a typical architectural solution to deal with complexity, maintainability and efficiency of FT mechanisms.

4.2.4 Architecture description languages

Architecture description languages (ADLs) are applied to describe the system architecture. An ADL is a language for specification and modelling of a system's conceptual architecture [104], including system components, interfaces, and configurations. ADLs are classified into three categories: box-and-line informal drawings, formal ADLs and Unified Modeling Language (UML) notations.

Box-and-line notations are criticised [105] for being informal since the model produced by these notations can be approached from different perspectives by different stakeholders. In addition, it is difficult to check that a system implementation corresponds to its architectural description. These limitations of box-and-line notation were overcome by introducing various formal ADLs [104] that represent an architecture in a formal way and are readable by humans and computers. Formal ADLs focus on different architecture elements, application domains and analysis techniques. However, they have some disadvantages, since there is no consensus on what should be represented by ADLs and most of them are targeted at a particular type of analysis. A general-purpose UML notation [106] is intended to deal with the problems inherent in box-and-line informal drawings and formal ADLs. UML provides a standard way to visualise a system's architectural blueprints in various diagrams. These diagrams are presented by structure diagrams (component and class diagrams), behaviour diagrams, and interaction diagrams.

UML component diagrams and plain English were applied to describe all the details of the HFT architecture with its elements and interactions between these elements.

4.3 HFT controller

The HFT controller is a central element of the HFT architecture. It coordinates system-wide FT strategies and distributes available resources among the system components. These resources can include computer hardware, energy and power. In addition, it reconfigures the system components using special reconfiguration interfaces if it detects that the system can operate faster or more reliably. These tasks are mainly performed with the assistance of the HFT agents, which obtain all required information from the system components by monitoring them and passing this information to the HFT controller. Moreover, the HFT controller initiates fault handling and reconfiguration of the entire system after detecting certain erroneous conditions and checking error rates. In this case,

apart from the HFT agents' help, the HFT controller utilises public interfaces of critical system components in order to adjust the reliability, quality of service, performance and energy consumption of the system. However, it should not be aware of the inner structure or encapsulated information of the system components because in this case it will be very complex for maintenance and understanding. This is the reason why the knowledge of the HFT controller about the system should be restricted by the general structure of the system and the performance characteristics and average resource utilisation requirements of the system components. The HFT controller has information about the dependencies between the HFT agents and system components. However, in all cases where the implementation details of the system components are required to get run-time information about the internal behaviour and to perform some action, the HFT agents are applied, since they are implemented with knowledge of the components' inner structure and are responsible for their area of crosscutting concern. In this and the following chapters it is assumed that a computer-based system with the HFT architecture has only one HFT controller. Although configurations with several HFT controllers are possible, that would require special coordination between the HFT controllers. Such configurations could be needed only for very complex systems. Thus, systems with more than one HFT controller are not considered in this thesis.

4.3.1 Internal structure of the HFT controller

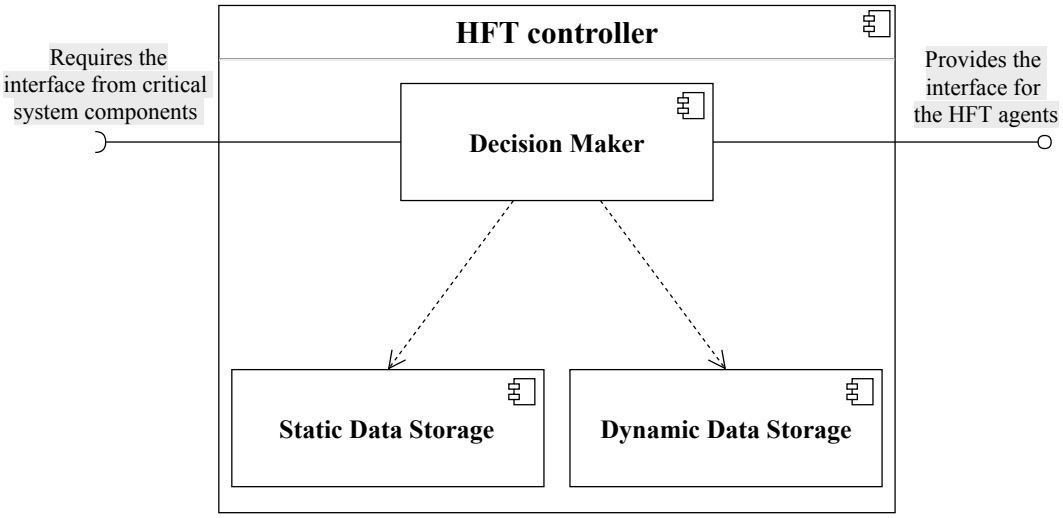

Figure 4.1: The internal structure of the HFT controller

The HFT controller consists of three parts: *static data storage*, *dynamic data storage* and *decision maker* (Figure 4.1). The static data represents predefined HFT policies that are mainly based on the system requirements. This includes expected system performance and

reliability, fault assumptions of the system (expected errors), available system resources, general structure of the system components and conditions for system reconfiguration. The dynamic data is the information about the current system state, which includes performance characteristics of the system components, error rates in critical functions and diagnostic information. These data are supplied by the HFT agents. The decision maker is responsible for reconfiguration and fault handling of the system. Moreover, it chooses the most suitable error recovery action for the error in the system component that should be handled holistically. These decisions are made based on static and dynamic data. The UML diagram in Figure 4.1 illustrates the HFT controller where the dynamic data storage is supplied with data from the decision maker, which filters the dynamic data received from the HFT agents. However, if all these data are useful (e.g. offline data analysis or machine learning) then it should be saved to the log file. In such a case it would be more convenient to assign this task to the dynamic data storage and implement an interaction between the dynamic data storage and the HFT agents directly without involvement of the decision maker.

4.3.2 Knowledge of the HFT controller

Not all errors should be handled with assistance of the HFT controller. A lot of errors should be handled locally by system components. The HFT modelling (Chapter 5) will provide precise information about the most suitable place for error handling.

4.4 HFT agent

The HFT agent is a special auxiliary object assisting the HFT controller. Each HFT agent is responsible for monitoring certain non-functional feature, such as performance or error handling in one or more system components. The HFT agent monitors and, if needed, intervenes in the control flow of activities in critical system components. This intervention should be as implicit as possible for the system components to avoid tangling of the HFT mechanisms and main functional responsibilities of the component.

The introduction of HFT agents is aimed at improving the scalability of the HFT architecture. The HFT agents are aware of the implementation details of critical system components and have the possibility and right of intervention inside the control flow activities of these components in order to check results, perform error detection and error

recovery, evaluate performance and suppress exception raising. The HFT agent is not aware of the entire system structure, because its goal is to monitor only parts of the system and pass the up-to-date information to the HFT controller. If the HFT agent monitors more than one component, errors could be detected based on concurrent analysis of two components. In this case, error recovery could affect both components as well.

4.4.1 Internal structure of the HFT agent

The HFT agent consists of *monitoring logic*, *intervention logic* and *local decision maker* (Figure 4.2). The monitoring logic defines which members (functions, properties or activities) of critical system components will be monitored by the agent. Monitoring does not involve any changes of state inside the system components. The intervention logic determines how the control flow inside monitored activities will be affected by HFT. The local decision maker of the HFT agent communicates with the HFT controller. The local decision maker distinguishes the data that should be transmitted to the HFT controller and the data that can be processed locally. These three parts of the HFT agent are application specific and they should be implemented depending on the type of the system, system requirements and an area of crosscutting concern. Some HFT agents do not implement intervention logic if they perform only monitoring actions, for example performance monitoring or function calls counter.

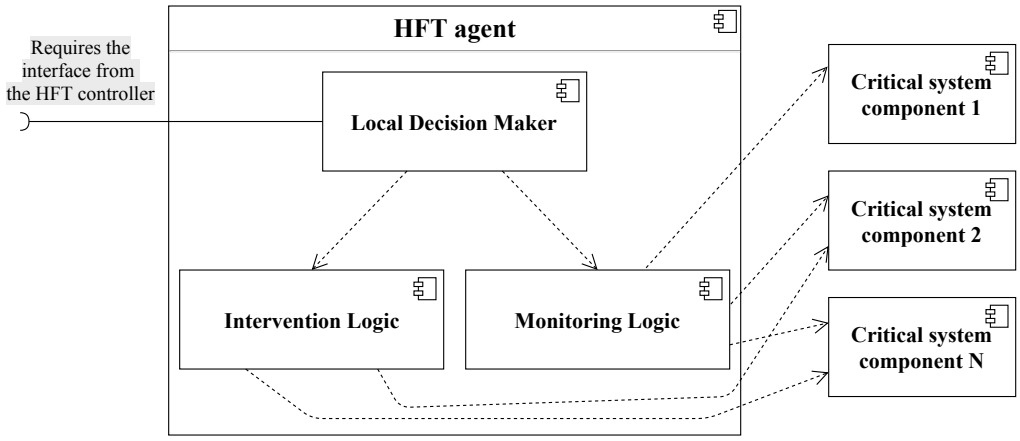

Figure 4.2: The internal structure of the HFT agent

4.4.2 Typical HFT agents

Various HFT agents can be applied in the system architecture depending on the type, size and requirements of the system. Typical examples of HFT agents include the Error

Handling agent, Performance agent and Diagnostic agent.

The Error Handling agent supports the HFT controller in implementing the system-wide FT mechanisms. It monitors the errors in critical system components that affect reliability of the entire system. When it is necessary the Error Handling agent intercepts the detected error and performs error handling in accordance with instruction from the HFT controller. Possible handling actions include error skipping, repeating erroneous operation (if the error is not permanent) or executing a stand-by alternate operation. It should be noted that some of these actions could be implemented as default error handling by the component itself.

The Performance agent monitors how fast the critical components are executing their tasks. If some activities or functions became slower than is expected from the component, the Performance agent informs the HFT controller that makes necessary system reconfiguration based on these data.

The Diagnostic agent is responsible for collecting diagnostic information from the system components. The obtained data are aggregated and passed to the HFT controller. The Diagnostic agent implements monitoring activities only and does not have an intervention logic since there is no need to alter the component behaviour.

In small- and medium-scale systems the responsibilities of the Diagnostic agent may be shared between the Error Handling agent and the Diagnostic agent when it is not practical to introduce a separate module for diagnostic activities and these activities can be done in parallel with error handling and performance monitoring activities. However, in bigger systems, it is more convenient to separate these activities and to introduce the Diagnostic agent.

Apart from these three agents, the design of a certain system could require other agents like Security agent, Resource Distribution agent or Energy Distribution agent.

The aforementioned recommendations act as a guidance and should not be followed extremely strictly. Thus, each implementation of the system with the HFT architecture will be unique and the choice of the HFT agents and their responsibilities could diverge from this guidance if in this case the system structure would be more rational and understandable.

4.5 Interaction between the HFT controller and the HFT agents

The HFT controller works with all available HFT agents. In case of error in an observed component, the HFT agent could request the HFT controller for a suitable error recovery. In addition, the HFT agent should detect the states in the monitored component that may cause an error, and propagate this information to the HFT controller. To reduce HFT controller complexity, it is advised to simplify the data sent by the HFT agents whenever it is possible. For instance, quality of an operation or result of a function could be presented by discrete enumeration: *Error*, *Low Quality*, *Medium Quality* and *High Quality*. The task to map from the component-specific data to the simplified data suitable for the HFT controller is performed by the HFT agents.

The HFT agents are intended to simplify the development and implementation of the HFT controller. The HFT agents get the information from monitored system components, transform it to the format suitable for the HFT controller and transmit this information. The data mapping to the HFT controller format is necessary to avoid tangling of the HFT controller with encapsulated details of critical system components. Otherwise, scalability of the HFT controller will be deteriorated. To improve performance and avoid bottlenecks in the HFT controller, the HFT agents should filter the information and send important data only.

4.6 Interaction of the HFT elements with the system

4.6.1 The HFT controller and system components

In the system with the HFT architecture, critical functional components should provide interfaces to the HFT controller. These interfaces will be used by the HFT controller for reconfiguration and fault handling. This is applied to deal with an interplay between reliability, performance and resource usage. In such a scheme the HFT controller does not need to know the implementation details of the system components, since it uses only a predefined interface and is aware only of general structure of the system. This link between the HFT controller and system components is supposed to be used only asynchronously. The decision maker of the HFT controller should operate in a separate thread (or process). Based on the information from the HFT agents, it can detect that

operation of the system could be more efficient or more reliable. In this case, the HFT controller specifies the most suitable configuration for the system component. If the system does not have any reconfiguration possibilities or various options of resource usage then the implementation of the HFT architecture would be much easier, however, in this case the HFT architecture will demonstrate maintainability benefits only. Reconfiguration is available for system components that provide some redundancy in their implementation. The concept of HFT implies that acceptable frequency and severity of the service failures could vary depending on user requirements or current system settings.

For the case of direct interaction between the HFT controller and the system components, the latter should provide special interfaces, which are used to adjust component settings, to choose operation mode and/or to perform fault handling by reconfiguring the component. These interfaces enable the HFT controller to tune the reliability and performance of the entire system. All actions that are not expressed via public interfaces and require amendment of the component behaviour should be done by the HFT agents implicitly to the components. Thus, the HFT controller does not have strong dependency on the system components, but it still has a global knowledge about the entire system.

In a software application such components (classes) should implement an interface by providing an implementation of methods for adjusting and reconfiguration (for instance `SetOperationMode`, `SetNumberOfWorkingThreads`, `SetQualityOfService`, `SetSettings` Java methods). An example of such an interface (`IAdjustableComponent`) is shown in Figure 4.3.

```
1 public interface IAdjustableComponent {
2   void SetOperationMode(...);
3   void SetNumberOfWorkingThreads(...);
4   void SetQualityOfService (...) ;
5   void SetSettings (...) ;
6 }
```

Figure 4.3: The interface for component reconfiguration

The FT mechanisms in the HFT architecture are distributed across the entire system, but coordinated centrally by the HFT controller. In some cases, it is beneficial to introduce redundancy in the FT mechanisms in such a way that the same error could be handled by the component itself and by the HFT agent. The decision on suitable error handling scenarios will be made by the HFT controller depending on the current system state. Such an approach provides flexibility in the choice of the optimal error recovery scenario.

Some errors will be handled by both the component itself and the HFT agent. Only part of the system components needs to be involved in the HFT mechanisms. It makes sense to use only critical components that globally affect the system operation or could be reconfigured in terms of performance or resource usage. It is definitely more convenient to implement these system components to be "HFT-ready" providing all necessary interfaces and preparing them to work with the HFT controller and the HFT agents. If the components do not provide such interfaces, for example legacy components, developers can implement special wrappers or adapters.

4.6.2 The HFT agents and system components

The HFT agents are aware of the inner structure and encapsulated implementation details of the critical system components, however, these components are implemented without knowledge about the HFT agents. On the one hand, this approach violates the abstraction and encapsulation principles because the HFT agent is significantly dependent on the structure of critical components. On the other hand, it assists in the separation of crosscutting concerns. It is not advised to use the HFT agents to amend the functional behaviour of the component. Instead, the HFT agents should be used to simplify the management of crosscutting concerns. Thus, the problem of implicit coupling between the HFT agent and critical system components is overlapped by better modularity allowing the developer to avoid code tangling and improve the understanding of system-wide FT techniques.

4.6.3 Operation modes

The HFT architecture supports operation modes by considering an interplay between reliability, performance and resource usage. Operation modes can be applied for the entire system and for the separate components. The HFT controller is the most suitable place to control and choose an efficient operation mode for the system components. Let us consider the following example. Two components are performing some operation. To finish one cycle, a chunk of data should be processed by one component and put in the queue. The second component checks the queue. When there is a chunk of data, it takes this chunk for processing. To balance the loading of the components we can specify how to distribute computer resources between these two components by operation mode assignation and how to balance the data chunk queue. This can be elegantly done by the

HFT controller. The HFT agents monitor these components and supply the information to the HFT controller, so that the HFT controller is able to increase or decrease the quality of service for each component on-the-fly. The operation mode can also be considered as a graceful degradation for the system when some component fails or requires a restart. This idea provides the possibility of fault handling with the assistance of the HFT controller, which performs system reconfiguration by choosing a suitable operation mode for the system components.

4.6.4 System with the HFT architecture

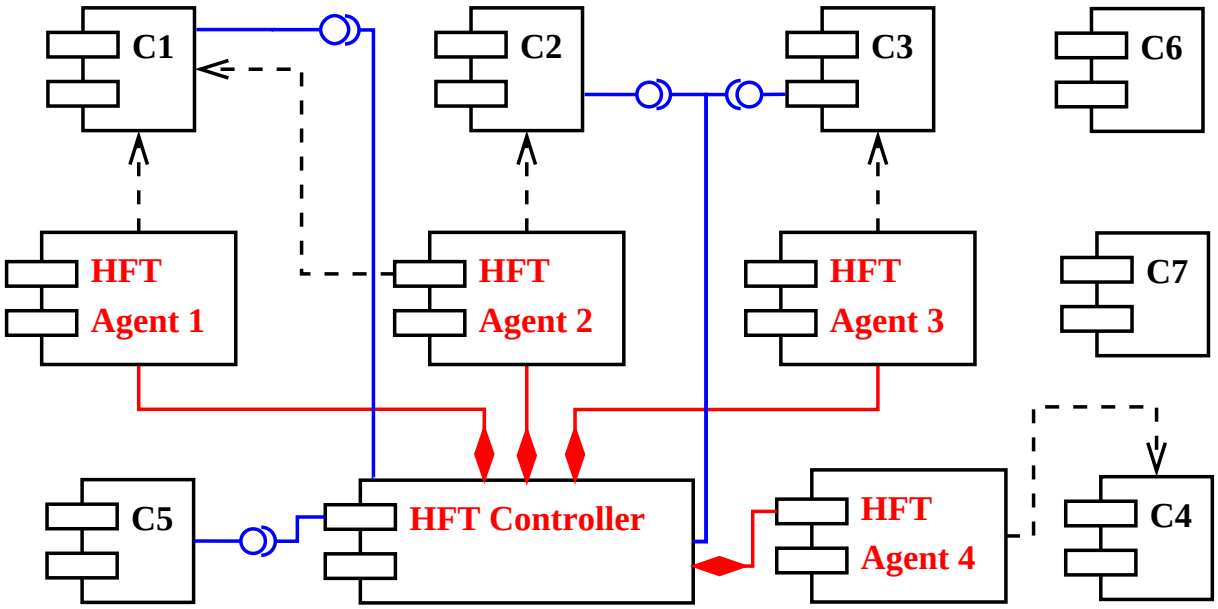

Figure 4.4: The system based on the HFT architecture

Figure 4.4 illustrates an architecture of a component-based system with HFT. This system has seven functional components (C1–C7) that implement functional requirements of the system and the HFT part (depicted in red colour) consisting of the HFT controller and four HFT agents. For the sake of simplicity connections between the functional components are omitted. The HFT architecture considers four groups of components depending on the way of interaction between these components and the HFT part. Components in the first group (C1, C2 and C3) are monitored by one or more HFT agents and provide the interface for the HFT controller. Thus, the given components can be used for the reconfiguration of the system and it is useful to monitor their inner operation to reason about the state of the entire system. The second group (C4) of components is only monitored by the HFT agent/s. The state of such components is useful for holistic monitoring, however, these components are not reconfigurable. The dependency relation between the HFT agents and

system components is implicit for the components. Thus, the system components do not know about the HFT agent. Components from the third group (C5) are not monitored by the HFT agents, but provide an interface for the HFT controller. Such components implement various operation modes and could be reconfigured when needed. However, it is impractical to monitor or intervene into their operation with HFT agents. The fourth group (C6 and C7) of components is not directly affected by the HFT architecture.

As is shown in Figure 4.4, components C1, C2, C3 and C5 provide interfaces for the HFT controller, while the HFT agents implicitly monitor and if necessary intervene in the control flow of components C1, C2, C3 and C4.

4.7 Discussion

The architecture of the HFT elements in different systems would be similar. However, their implementation significantly depends on the system being developed. Thus, the implementation of HFT varies from system to system.

It is impractical to connect all system components to the HFT part. It is necessary to choose only those components, which affect the operation of the entire system and are able to provide important information about the system state. If some of these carefully selected critical components are already implemented or reused then there is less flexibility to use them in the HFT mechanisms without additional elaboration. However, if these components are not implemented yet or there are no obstacles preventing their modification then it is reasonable to make these components HFT-ready. This includes introducing some redundancy to support different operation modes and reconfigurations.

The responsibilities and structure of the HFT controller and the HFT agents should be defined according to the system requirements, fault assumptions and application domain of the system. This is followed by the description of the interactions between the HFT controller, the HFT agents, and critical system components. The interactions include reconfiguration interfaces for the HFT controller, monitoring and intervention mechanisms for the HFT agents, and the information flows between the HFT elements and functional system components.

After the first prototype of the architecture has been created during the design step the system architecture may require refinement and evolution to ensure its efficiency. The HFT modelling step described in Chapter 5 is intended to ensure that the system based

on the HFT architecture will be efficient.

An important question regarding this architecture is how to avoid a single point of failure, since FT strategies and system reconfigurations are managed by the centralised HFT controller. One of the options is to ensure dependability of the HFT controller and the HFT agents using standard FT techniques [55]. The complexity of this implementation depends on system criticality. Apart from this option, the "default" system behaviour for the case when the HFT controller or one of the agents fails should be implemented by those system components that are involved in the system-wide HFT mechanisms. In this case the system would operate not optimally and possibly less reliably, but system failure will be avoided and the service will still be provided. The main thing is detection of the problems with the HFT elements and well-timed correction to return the system to efficient operation. However, it does not change the fact that the HFT mechanisms should be developed rigorously and they should be dependable.

4.8 Conclusion

This chapter presented an architectural pattern for the design of computer-based systems with the HFT architecture. The guidance and advice on how to structure and architect the system with HFT were provided. A typical system based on the HFT architecture consists of functional components satisfying functional requirements and the HFT part, which is responsible for dependable and efficient operation of the system. The HFT part includes one HFT controller and several HFT agents. Critical system components are implicitly monitored by the HFT agents, and, when necessary, reconfigured by the HFT controller through special reconfiguration interfaces. However, there is a need to ensure that the proposed architecture allows the system to be dependable and operate efficiently. In addition, apart from the design, the implementation stage of HFT engineering should be described.

The HFT modelling intended to guarantee the efficiency of the system designed in accordance with the HFT architecture will be presented in the next chapter. After this, guidance on the implementation of the software application with the HFT architecture will be provided in Chapter 6.

5

Modelling of computer-based systems with Holistic Fault Tolerance

Contents

In the two previous chapters we described an engineering approach to designing computer-based systems with HFT and provided a thorough description of the HFT architecture. In this chapter, a method to assist in modelling the computer-based systems with HFT is proposed. The method is intended to refine the interactions between the HFT part and critical components of the system designed at the previous step (Chapter 4). This novel HFT design method is based on hierarchical modelling and stochastic simulations. The former caters for system complexity whereas the latter supports reasoning about non-functional properties and their trade-offs. This chapter starts by introducing the general modelling method and demonstrates its applicability with a case study of a number plate recognition application.

The chapter is organised as follows. A brief introduction to the chapter is given in Section 5.1. The background material on Control Theory, Stochastic Activity Networks and Order Graphs, and the reasoning behind the choices of the modelling approach are given in Section 5.2. The aims of the modelling are discussed in Section 5.3. The algorithm for the creation of Order Graphs models and the steps of the modelling method proposed are presented in Section 5.4. The practical usage of this modelling method is demonstrated in Section 5.5 using the case study. Section 5.6 briefly summarises the work presented in the chapter and draws some conclusions.

5.1 Introduction

System modelling aims to create an abstract representation of the system under design. This process assists in a better understanding of the system and its structure and gives a possibility to find and eliminate potential problems at early stages of system development. However, for complex systems the system model can also be complex and difficult to use. Therefore, it is necessary to ensure that only important parts of the system are studied during modelling to reduce comprehension complexity. By the important parts we mean those components of the computer-based system which are responsible for the main system functionality and significantly affect the system characteristics, such as quality of service, performance and reliability.

A widely-accepted method for dealing with model complexity is the use of hierarchical models. Different models may be constructed for the same system or a subsystem at different levels of abstraction. On the one hand, high-level models of high degrees of

abstraction tend to be smaller and easier to analyse, but also include few details and may provide low representative resolution or precision for the characteristics or the parameters being studied [107]. On the other hand, less abstract low-level models may offer a finer grain representation of system detail and provide a higher precision for modelling the parameters of interest. However, they may have high degrees of complexity and be more difficult to work with. Hierarchical modelling provides designers with a means of trading off modelling quality with model usability.

Another popular method of dealing with model complexity and at the same time handling run-time unpredictability is stochastic modelling. Quantities and parameters under study are assumed to be of a stochastic nature, whereas the models of a manageable size can be used to estimate such quantities without precise knowledge of all the contributing factors such as run-time eventualities [108].

Aspects of study during system design and analysis include functional behaviour and non-functional parameters. Functional correctness is important, but non-functional parameters can also be significant contributors to the success or failure of a design. The most interesting non-functional parameters attracting the attention of system designers include reliability, performance and resource usage. Depending on the application domain of the computer-based system, resource usage can be presented by energy consumption, power consumption, hardware resource allocation or by a combination of these characteristics.

As presented in Chapter 4, the HFT architecture includes system components, an HFT controller and a number of agents which supports interactions between the components and the HFT controller. The HFT run-time is implemented through control loops that manage the non-functional parameters through component configuration. However, there remain challenges faced by the HFT developer during the design stage. It is not always clear how to choose the system components which will be involved in the interaction with the elements of the HFT architecture (essentially the number of control loops). If the designer chooses to involve all system components in the interaction with the HFT elements, i.e. have the maximum number of all possible control loops included, the system would be extremely complex for modelling, implementation and maintenance. On the other hand, unguided control loop reduction would rarely result in optimal system design.

This chapter focuses on modelling that supports design-time and run-time system optimisation through the (re)configuration of system components and the efficient use of control loops. However, the model should not be very complex for understanding. Iterative top-

down design and stochastic representation of non-functional parameters offer promising solutions.

A general design method supporting HFT systems is proposed. This method makes use of a hierarchical model language, known as Order Graphs (OGs) [95], which has good representations of horizontality and verticality issues and good support for having different levels of abstraction for different parts of a system model. Also included is an established stochastic model language, known as SANs [108], which provides facilities such as state-space analysis and simulation engines.

The proposed design workflow is based on the following key points:

- The characterisation of system components leading to SANs models. These SANs models can be used to provide estimates of the non-functional parameters under study (usually reliability, system utilisation and/or performance) and generate importance costs for potential control loops in the HFT control.
- The concept of controllability is applied to minimise the number of control loops.
- The development of a hierarchical model of the HFT system based on OGs. This model can be used to validate the existence of control loop paths at all levels of model abstraction.

5.2 Background

In this section, the underlying background material covering Control Theory, SANs and Stochastic modelling and OGs modelling, is considered. It is important since HFT modelling is based on these areas.

5.2.1 Control Theory

During the modelling the concepts of Control Theory, such as transfer functions, control loops and typical steps applied for control problems are referred.

Control Theory [109, 110] is applied to control continuously operating dynamic systems. Control Theory is presented by classical control theory and modern control theory. Classical control theory is suitable only for single-input, single-output systems, which is not

enough for complex modern systems. Modern control theory, in turn works with multiple-input, multiple-output transfer functions and is based on time-domain analysis of differential equation systems. It made the design of modern control systems much simpler.

The *transfer function* in control theory is a mathematical representation of the relation between the input and output of the component in a form of linear, time-invariant or differential equations that are applied to describe the system.

A *controller* in control theory is a special device that monitors and alters the operating conditions of the controlled dynamic system.

There are two types of control loops: *open loop control* and *closed loop control*. In the former case, the controller does not analyse the system output. Sometimes open loop control is called feed-forward control. The latter is called feedback control and in this case the controller's action depends on the system output. In the system with the HFT architecture both types of control loops are applied. Open loop control is between the HFT controller and system components. The HFT controller decides whether it is necessary to make a certain reconfiguration of the system. Closed control loop exists between the HFT agents and system components, and between the HFT agents and HFT controller.

A significant issue during the system modelling is a trade-off between simplicity and accuracy. Simple models could be inaccurate, whereas precise models could be very complex for comprehension.

There are three main steps that are applied to the majority of control problems:

- building a mathematical model that is a simplified representation of a real system.
- applying mathematical analysis and design techniques to the model.
- interpretation of the mathematical results.

5.2.2 SANs and Stochastic modelling

SANs [111, 112] is an extension to general stochastic Petri nets (GSPNs) which are based on Petri nets (PNs) [113]. It inherits the general attributes of PNs including a distributed representation of system states, making it easy to represent parts of a system directly as local subsystems, and more straightforward representations of such important issues as concurrency and synchronisation. A well-established method, it is supported by various software modelling tool-kits like METASAN [114], UltraSAN [115], and Möbius [116, 117].

SANs are capable of representing both deterministic and stochastic events, and event durations in time. The elements used in this work include a) transitions whose firing speeds (rates) are specified as stochastic, following given distributions, and b) transitions with multiple firing cases with specific probabilities for each case.

The Möbius tool was applied because it provides all the features required for SANs modelling and it is supported by most modern operating systems. The Möbius tool, incorporates a set of solvers including both Monte-Carlo simulation and state space related solvers. Numerical Markovian solutions can be done for steady-state or time averaged interval rewards, but are limited to models with exponentially distributed firing rates. The tool concept of "rewards" is used to evaluate a system's non-functional properties including performance, reliability (defined as success rate), and resource utilisation.

5.2.3 Order Graphs and resource modelling

OGs [95] is a system model with a hierarchical structure, which provides zooming across several layers of detail (orders). What is more important, zooming can be performed independently in different parts of the model, thus allowing the designer to make cuts through different orders. This is convenient when there is a need to focus on a particular parameter whose values are commensurable only at different orders (levels of detail) in various parts of the model. However, these parameters are significantly different at the same order. For example, if there are two components, the first is responsible for intensive calculations, whereas the second stores application settings, then the CPU usage of these two components would be significantly different. If there is a need to analyse the CPU usage of these components in the context of a system, then it is more practical to consider subcomponents of the first component and the entire second component. Only in this case would it be possible to analyse parameter values that are not significantly different. This is impossible if the analysis is done at the same level of detail.

The aforementioned cuts can be analysed as a flat model during design time and run-time. This significance-driven modelling facilitates the modelling process and allows the designer to consider only those parts of the model and corresponding parameter values that are really important for the chosen modelling task without overcomplicating the model.

OGs [95] have been chosen because this technique allows us to model complex systems at the required level of detail. For non-important components the designer can stop earlier and not put any effort into the detailed characterisation of the components. Various

approaches, such as discrete event simulation [118] and Petri nets [113] were considered as candidates for HFT modelling, however all of them lack hierarchical modelling and possibility to create cross-layer cuts. These features are available for OGs and they are vital for analysing an interplay between reliability, performance and resource utilisation of various components in the system at different level of detail.

Hierarchical representations have been used for modelling complex systems for a long time. The idea of separating the "vertical" relation between the layers of abstraction from the "horizontal" knowledge of the system at each particular layer of abstraction has been hinted at in [119] and then formally defined in Zoom structures [120] as the concepts of verticality and horizontality. Zoom structures are based on partial orders and are very permissive. In contrast, OGs put a number of constraints on the modelling, which guarantee consistency between the abstraction layers.

An Order Graph (OG) is a graph with nodes representing various system resources, such as hardware components, energy or power budget, arranged in tree hierarchies. The hierarchies can be built from the knowledge of the system structure and by similarities of its constituents. The distance from the root relates to the level of abstraction. The formal definition and properties can be found in [95].

Modelling using OGs is an iterative top-down process, starting from the most abstract representation of the system and gradually adding more detail, when moving to lower levels. The dependencies between the system's components at the same level of abstraction are represented with "horizontal" arcs in the graph, hence the horizontal paths represent transitive dependencies between the elements in the system. The rigorous definition of OGs provides a built-in capability of consistency checking by preserving the resource dependency paths at each level of abstraction. For small and medium scale systems OGs modelling can be done manually as is shown in Subsection 5.4.5. However, there is an ArchOn [121] modelling tool for OGs that is currently under development by the PRiME project team at Newcastle University.

5.3 Aims of the HFT modelling

Modelling allows us to design an abstract structure of the system and focus on problematic points at early stages of system design. Ideally, the majority of possible flaws and logical inconsistencies should be eliminated at this step. In addition, it should allow to select

only those interfaces between the critical system components and the HFT part that are really important for reliable and efficient system operation.

In supporting HFT it is proposed to use modelling at that step of the system engineering when the majority of the critical system components are ready or at least they have full functional characterisation. Outcomes of the modelling process will assist in selecting the correct set of interfaces for the interaction between critical components and the HFT part. It allows the designer to remove redundant interfaces and to add missing interfaces to achieve efficient operation. This is especially crucial for the computer system, which is able to operate in various operation modes. The most important outcome of the modelling is the ability to design an effective HFT for the system. Modelling answers questions about the control loops that should be implemented between the HFT part and functional part of the system. Unnecessary control loops, which do not provide benefits for system efficiency will be defined and eliminated during the modelling step. Thus, HFT modelling allows the designer to model the computer-based system, which will be efficient with regards to the chosen non-functional requirements.

HFT modelling should assist in design and implementation of the system, which will operate as close as possible to Pareto efficiency [122] state. Pareto efficiency is such a state of the system when it is impossible to improve one criterion without deteriorating the others. With regards to HFT, these criteria are abovementioned non-functional parameters: reliability, performance and resource utilisation. Sometimes, all these criteria can be improved by reducing the level of abstraction and, as a consequence, by deteriorating understandability of the system. However, in this case, it makes sense to consider system abstraction as a criterion of Pareto efficiency as well.

5.4 Modelling method

In this section, the workflow of the proposed modelling approach is explained. In addition, non-functional properties of a computer-based system under development are defined. These non-functional properties are of interest for HFT, since it is necessary to define an effective way to monitor and adjust the system to ensure efficient operation.

The goal of modelling is to provide a method that allows the developer to design and implement an efficient system based on the HFT architecture. It is necessary to guarantee that the system will be efficient with regards to non-functional properties, such as relia-

bility, performance and resource utilisation. Modelling assists in defining efficient points of the interplay between these non-functional properties. An *efficient design* allows the developer to implement such a system, which will be efficient in terms of this interplay.

Performance is considered as the amount of work completed per unit time. Faster operation typically requires more resources or can be achieved by reducing the quality of computation. The work performed by the system is measured in work units. The processing of each work unit can be finished successfully or unsuccessfully.

Reliability is represented with the success rate, which is defined as the ratio of successfully finished work units to the total amount of work units.

Resource utilisation is the amount of computer resources required to process a certain number of work units. In this context, a resource is defined as any facility that enables computation, which may include CPU cores, application threads, memory, energy, etc.

A computer-based system with the HFT architecture contains functional components that provide the main system functionality and the HFT part (Chapter 4). The HFT control for the non-functional properties is realised using *knobs* and *monitors*. The knobs are provided by system components as configuration points (or interfaces for the HFT controller), and the monitors are an instrumentation that provides readings of values and parameters at the component level. These readings performed by the HFT agents assist in reasoning about the non-functional characteristics of interest. The system-wide set of knob states is called a *system configuration*. During the system operation, the HFT controller dynamically choses the most suitable system configuration, depending on the history of observed data.

An instance of such interaction between the HFT elements and the functional system components is defined as a *control loop*. It can be considered as a special interface between the components and the HFT part. The control loops are managed only by the HFT part and they can be either explicit or implicit to the system components. The former control loops are presented by the reconfiguration interfaces used by the HFT controller, whereas the latter define the interaction between the HFT agents and the system components.

5.4.1 Workflow of the modelling approach

The workflow of the HFT system modelling approach is described in Figure 5.1. Each of the steps is explained in a subsequent subsection. Note that OG modelling happens

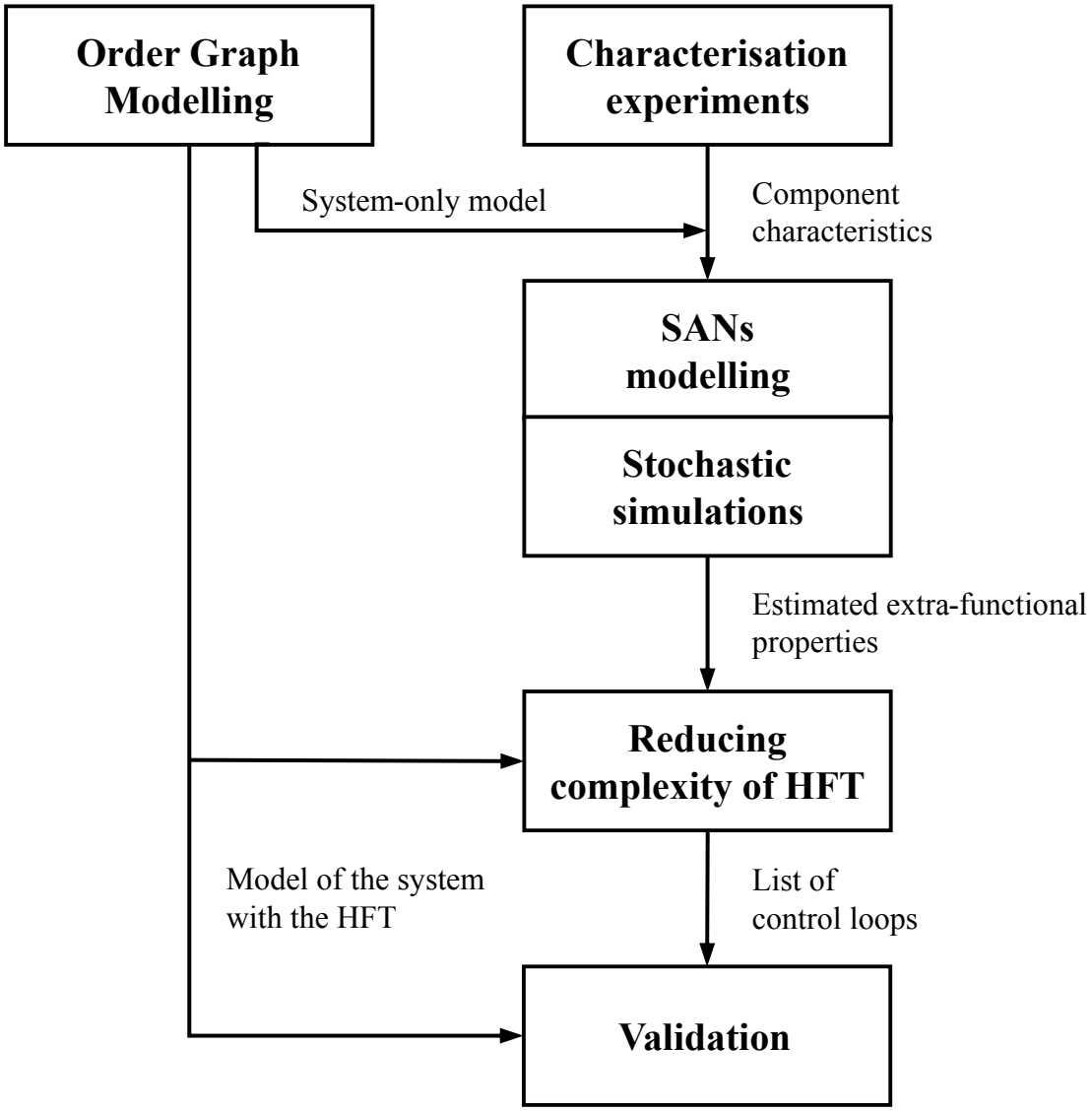

Figure 5.1: Workflow diagram

in parallel with the main branch of the workflow. The inputs of the modelling include characterisation of the critical system components, Stochastic Activity Network (SAN) model and OG model. The output of the modelling is a validated OG model of the systems, which allows the designer to select only important control loops between the HFT part and critical system components and ensure that there is no redundant interactions between them.

5.4.2 Characterisation of the component non-functional properties

Characterisation is the process of finding out how a system behaves with regards to non-functional properties by running the system under carefully selected input patterns and extracting the properties from a set of runs. The results of characterisation leads to

models, the running of which simulates system behaviour under input patterns that have not been experimented in characterisation. For instance, at the end of the characterisation process it would be possible to say what is expected performance or reliability of one of the critical components over a broad range of input and operating conditions. Then these expectations are used in the analysis, but not the actual values obtained from the experimental runs.

The designer should characterise the non-functional properties of each critical component which is involved in the HFT mechanisms (Section 4.6). If the component supports multiple configurations or algorithms of processing, the characterisation should be done for each individual configuration. First of all, to characterise the component, it is necessary to choose several input data sets, which are typical for the given component and are expected to be processed by the component. The choice depends on many factors. Depending on the system complexity and required model accuracy the number and type of input data sets could vary. It therefore is of vital importance to correctly select the input patterns and other conditions for characterisation experiments in order that the resultant data is meaningful for the type of models being built. If there is a need to model an "extreme" behaviour of the component, when unusual input is given to the component, then input data sets should contain non-typical data. Otherwise, the input data should be chosen according to the principle, when it is as close to real conditions as possible, while spending a reasonable amount of time for input data sets creation.

Then, each data set should be processed by the component. Finally, after the processing, the component characterisation can be made. This characterisation includes expected performance, expected reliability and expected resource utilisation for processed data sets. The full result of a characterisation pertaining to some component and some non-functional property describes the value of that property when executing that component. Characterisation is not done beyond the component level.

5.4.3 Building and simulating the SAN model of the system

This step defines how the HFT part (the HFT controller and the HFT agents) should be designed and implemented. The decision on structure and responsibility of the HFT elements will be made based on the simulation results. The SAN model includes critical system components, however the HFT elements are not in the model. Modelling of the components responsible for HFT is a part of future work, which is related to adaptive

HFT (Subsection 8.5.1). By simulation we assume running models and not the systems themselves to derive quantitative or qualitative expectations of non-functional properties.

In this step the SAN model of the system is built using component characterisations from the previous step and the system-only OG model (Subsection 5.4.5). The granularity of the SAN model for any part of the system is determined by the OG modelling step (Figure 5.1) and the parameter values are obtained from the characterisation step. The characterisation step usually pertains to the SAN model of the finest detail, because there is no point in developing a SAN model at a finer level of detail than the existing characterisation data. If the OG step suggests a higher level of abstraction, it is possible to derive SAN models of less detail than the characterisation data, for instance by running simulations at the characterisation level of detail then abstracting from the results.

From characterisation to the final SAN model for simulations the approach is bottom-up, but the OG step is usually top-down. There is no conflict because in order to determine the granularity of the final SAN model the entire OG model covering all levels of abstraction needs to have been established. In a way discovering the SAN model is a process of raising the level of abstraction from the bottom traversing the OG until a satisfactory SAN has been found.

The preferred tool for working with SANs is Möbius [117]. The main point of this step is that the SAN system model, assembled from component models, supports system-wide analysis of the modelled non-functional properties from component-level characterisation data. The most practical analysis method for SAN models of HFT is simulation, as other forms of analysis such as state space studies tend to be restricted to very small models. However, Möbius does provide non-simulation solvers if and when they can and need to be used.

5.4.4 Reducing complexity of HFT

The estimated values of system-wide non-functional properties obtained from the previous step can be used to reduce the complexity of the HFT controller by eliminating unnecessary control loops (control loop pruning process).

The method is based on the problem of preserving controllability [109] while reducing the number of knobs. It assumes that the number of monitors is both sufficient and necessary to represent the non-functional properties under study. The monitor values are considered state variables.

Möbius simulations (running the model that was created in the Möbius tool) are used to built a system transfer function [109] relating knobs to monitors. This is achieved by analysing differentials in the estimated monitor values from simulations. Ideally, this requires an exhaustive set of simulations covering all combinations of knob values. For the systems that require high precision of simulation and tuning it is advised to apply other known optimisation methods, such as Monte-Carlo [123], to improve the usability of the method.

The smallest set of knobs that maintains controllability is determined from this database of state relations.

Although in this chapter we deal only with deciding what control loops to include in an HFT system, the off-line design flow described here can yield valuable quantitative data that may be helpful for the detailed design of run-time control. For instance, the SAN models may provide a set of reference points which may be used in the design of individual control loops.

5.4.5 Validation using Order Graphs hierarchy

OGs modelling proves that control loops between the HFT parts and the system components are sufficient and not redundant. A rigorous path consistency checking between the layers of abstraction guarantees that the designed HFT part is consistent with the control loops established in the previous steps of the workflow.

As mentioned in Section 5.2, OG modelling provides a top-down workflow that helps the developer to incrementally add the details in the system design. In the proposed workload, the dependencies in the graph represent interactions between the elements of the system and provide paths for the control loops.

OG contains the static knowledge of the system and needs to be paired with a dynamic model to capture the system behaviour (in our case: SANs). The nodes in OG that are included in this model relation form a cut. If the cut goes through different depths in the hierarchy (layers of abstraction), it is called a cross-layer cut. The cut containing all leaves relates to the most concrete (detail) model of the system. Moving up in the abstraction hierarchy, thus grouping multiple nodes into one, represents grouping the corresponding elements in SANs into a single entity by averaging/totalling their parameters (known as black-boxing). This reduces the size of a model, but also introduces inaccuracy. The

trade-off between model complexity and accuracy can be achieved from manipulating cross-layer cuts. This method, called selective abstraction, has been explored in detail in [107].

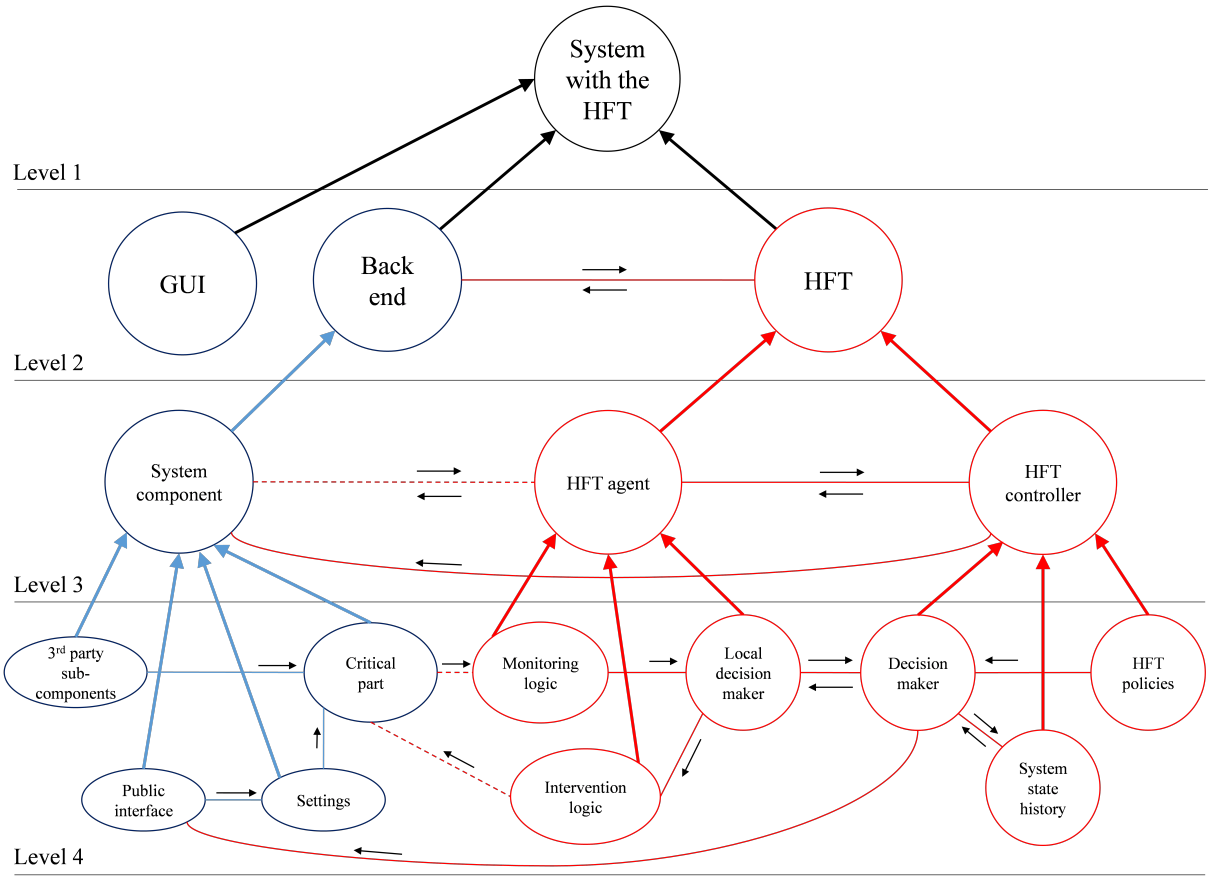

Figure 5.2: A general template for HFT Order Graph model

Figure 5.2 illustrates the hierarchical model of a computer-based system with the HFT architecture (Chapter 4) in three levels of detail. At the top level, there is only the system with the HFT architecture. The second level contains graphic user interface, backend or functional part of the system and the HFT part. Information flow between the backend and the HFT is shown in both directions. The next level represents the backend is decomposed to the system components and the HFT part decomposed to the HFT controller and HFT agents. For simplicity, the figure shows only one component and only one agent. At this level, the control loops between the system components and the HFT elements should start to appear.

The most detailed level of the hierarchical model considers the inner structure of the system components and the HFT elements. The system component may include third-party subcomponents, public interfaces, component settings and critical parts. The internal structure of an HFT agent (presented in Subsection 4.4.1) consists of monitoring logic,

intervention logic and local decision maker. The HFT controller (Subsection 4.3.1) includes the decision maker, the system state history (dynamic data storage) and the HFT policies (static data storage).

Connections represented by the dashed lines assume that for better maintainability it is preferable to implement this link in such a way that the system component is not aware of implementation details of monitoring and intervention logic in the HFT agent. HFT agents do not directly provide performance of reliability benefits. They were introduced to simplify the developing and improve understanding of the systems with HFT. It will be shown in Section 7.4 that such configuration supports maintainability of the system. This is the reason for decomposition of the HFT architecture to the HFT controller and the HFT agents.

5.5 Modelling of the case study

5.5.1 Case study application

The application for the recognition of UK number plates was chosen as a use case. The input of the application is a set of images. As an output, the application links each image with recognition results that include the contour of the number plate, recognised string and the probability of correct recognition.

The functional part of the application consists of several components. The Graphical User Interface (GUI) component is the frontend of the application, which allows the user to upload the images. These images are sent to the Initial Image Processing (IIP) component. At this stage, every image undergoes an initial processing, which includes various filters, searching of the number plate on the image, cropping of the number plate from the image and elimination of the perspective skews of the number plate cutout. Two algorithms for number plate search can be applied: OpenCV-based [124] rectangle detection and HAAR cascade [125] trained to recognise the area with the UK number plate. If the number plate is found and cropped it is put to the Number Plates Queue (NPQ). When the NPQ is not empty, the Optical Character Recognition (OCR) component takes an available number plate cutout and performs text recognition on the cutout. There are two OCR algorithms in the OCR component: Tesseract [126] and the number plate recognition algorithm described in [127]. If the OCR recognises the text on the cutout, this text is checked by the Result Checker (RC) component to ensure compliance of the car number

with a national format. These additional algorithms are introduced to provide redundancy and increase reliability of the application.

The UML diagram of the application is shown in Figure 5.3. The GUI does not participate in the HFT scheme and it should not be considered in detail in the model. Interfaces between the functional components (IIP and OCR) and the HFT controller are omitted to make the diagram clearer.

In both IIP and OCR components the images are processed concurrently. The HFT controller specifies the most suitable number of working threads for each component. The Performance agent monitors the execution time of the IIP and OCR components. The Error Handling agent is responsible for handling the errors in the IIP and OCR component. An error implies a deviation from the correct service [1] and it is not necessarily an exception only. Impossibility to find the number plate or low probability of the recognition is considered as an error as well. At the same time, not all exceptions are regarded as errors. If the error is detected by the Error Handling agent, it requests the HFT controller for a suitable error recovery action, which could vary depending on current system operation. A more detailed description of the application can be found in Section 7.2.

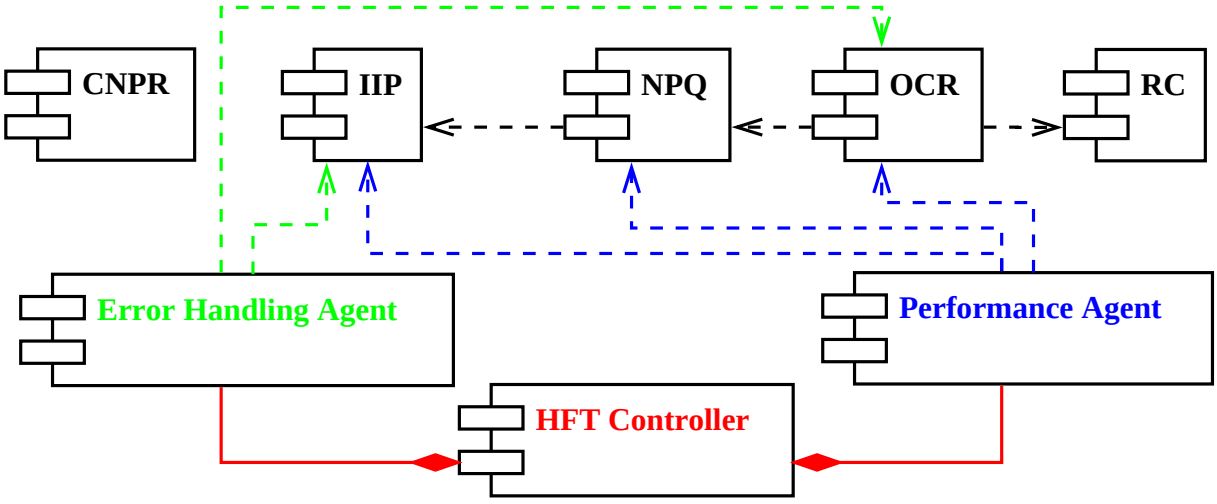

Figure 5.3: The UML diagram of the use case application

5.5.2 Characterisation of the components

For the characterisation, the IIP and the OCR components were chosen, since they are the most critical components of the application. Characterisation data is presented in Table 5.1 and Table 5.2. The input data varies significantly for the given application,

hence three groups of images distinguished by size have been chosen: small, medium and large. Time and reliability of the image processing significantly depends on the image size.

Table 5.1: Characterisation of the IIP component

Original image size		Number plate detection algorithm			
		Rectangle detection		HAAR cascade	
		Average time	Average reliability	Average time	Average reliability
Small	< 200 KB	20 ms	85%	9.3 ms	77%
Medium	200 KB – 1MB	85 ms	80%	76 ms	85%
Large	1 MB – 7 MB	143 ms	72%	328 ms	86%

Table 5.2: Characterisation of the OCR component

Original image size		Optical Character Recognition algorithm			
		OpenCV implementation		Tesseract	
		Average time	Average reliability	Average time	Average reliability
Small	< 200 KB	23 ms	70%	33 ms	75%
Medium	200 KB – 1MB	29 ms	73%	37 ms	78%
Large	1 MB – 7 MB	45 ms	48%	50 ms	62%

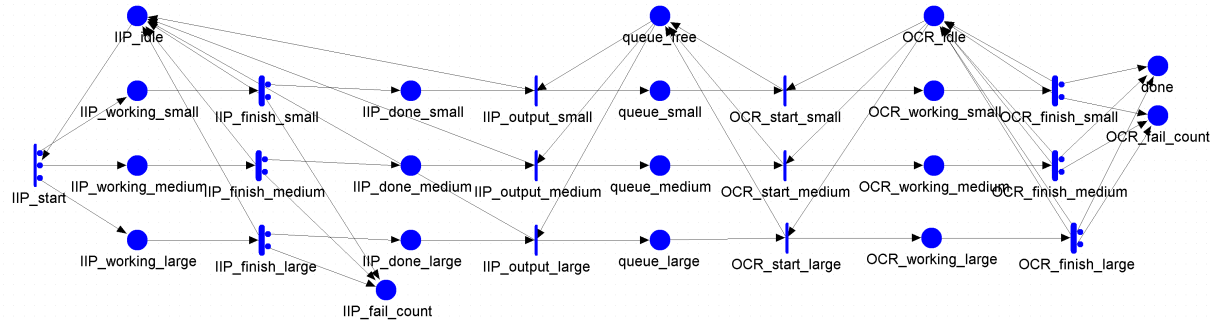

Figure 5.4: Detailed SANs model of the use case application in Möbius

5.5.3 SAN modelling and simulations of the system

With this characterisation data, the SAN models can be built in Möbius. A detailed SAN model for the two components IIP and OCR, each in three versions small, medium and large is shown in Figure 5.4. These component versions can be considered as thread pools for processing of images of corresponding size. The fundamental states for each component version are working and idle. Working means that this component version is in execution and idle means that it is not in execution. The model is simplified to put all idles together. This means that for, e.g. IIP, the IIP_idle place is initialised with the number of threads given to the IIP component. This may be known as the IIP capacity of the system. Each completion of an IIP component version puts a token back to this idle place. Each IIP component version has a probability of success P_s and a probability of failure $1 - P_s$ and this is represented by the stochastic timed transitions IIP_finish. The OCR component models have the same structure. Between the IIP and OCR blocks, three queues are modelled with the standard SAN representation for queues or buffers. The IIP_start transition on the left generates input images stochastically according to probability functions and rates that can be set in the model.

The occurrences of failure are tracked by the markings of the failure places and the overall number of successful recognitions is recorded in the final done place at the right end of the net. Running simulations with this model produces success and failure rates, resource utilisation (e.g. the average number of threads being active) and overall performance.

This model turns out to require somewhat significant time (more than a few minutes) to simulate. As a result, by making OG analysis and studying the characterisation data, a reduced model was derived, which is shown in Figure 5.5.

The reduced model only has a single OCR component version by combining the three different versions in the detailed model into one using average behaviour. The reason

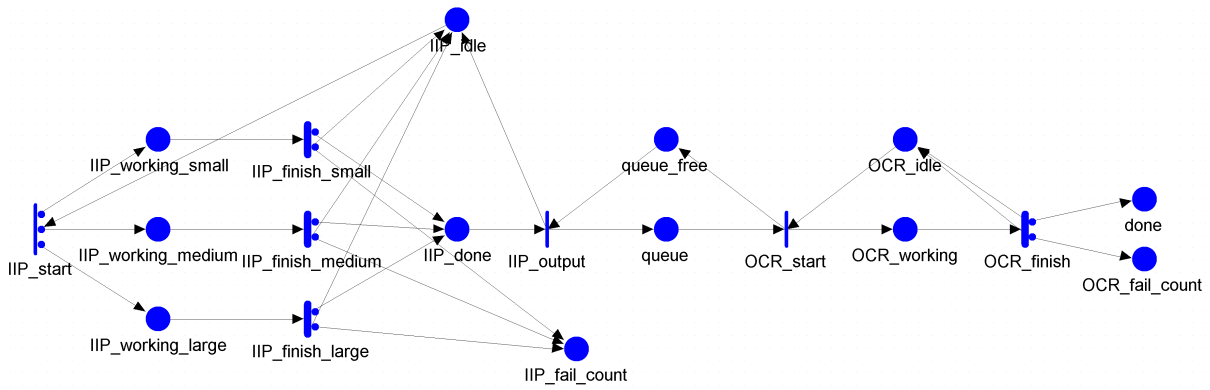

Figure 5.5: Reduced SANs model of the use case application in Möbius

behind this is that the version pertaining to large size is significantly slower than the others, which are very similar. Intuitively, component capacities are used more on the faster processing versions as they tend to grab the token from the idle place more frequently.

The reduced model required simulation times that are an order of magnitude shorter than the detailed model, and they produced very close results with differences within 5% on all the non-functional properties being studied. Some simulation results are shown in Table 5.3.

Table 5.3: Simulation results

Configuration				Estimates		
IIP algorithm	IIP threads	OCR algorithm	OCR threads	Core allocation	Success rate	Image time (ms)
Rectangle	4	OpenCV	4	4.55	0.543	44.89
Rectangle	2	OpenCV	6	2.4	0.55	55.91
Rectangle	6	OpenCV	2	6.63	0.548	40.77
Rectangle	1	OpenCV	7	1.24	0.528	87.11
Rectangle	7	OpenCV	1	7.49	0.559	40.68
Rectangle	1	OpenCV	1	1.22	0.529	88.26
Rectangle	3	OpenCV	1	3.4	0.553	48.86
HAAR	4	Tesseract	4	4.53	0.574	90.54
HAAR	2	Tesseract	6	2.36	0.577	116.89
HAAR	6	Tesseract	2	6.57	0.568	77.46
Rectangle	4	Tesseract	4	5.42	0.561	46.23
Rectangle	2	Tesseract	6	2.86	0.564	56.53
Rectangle	6	Tesseract	2	7.27	0.551	41.91
HAAR	4	OpenCV	4	4.26	0.561	90.01
HAAR	2	OpenCV	6	2.2	0.589	117.79
HAAR	6	OpenCV	2	6.31	0.557	76.98

In these particular simulations, we intended to find out if the relative numbers of IIP and

OCR threads executed affect the execution time, resource utilisation and reliability. It was found that the reliability stays about the same, but running more IIP threads than OCR threads improved the overall execution time and resource utilisation (more threads get executed simultaneously, pressing more cores and reducing idle time and queue length).

If the observed change in reliability is considered insignificant, the reduction of control loops leads to removal of all knobs except the number of IIP threads. This remaining knob provides control over resource utilisation and performance. On the other hand, if the reliability difference is considered significant, all knobs contribute to controlling the system properties.

5.5.4 Hierarchical model of the system

A hierarchical model of the system is built following the general template (Figure 5.2) and is shown in Figure 5.6.

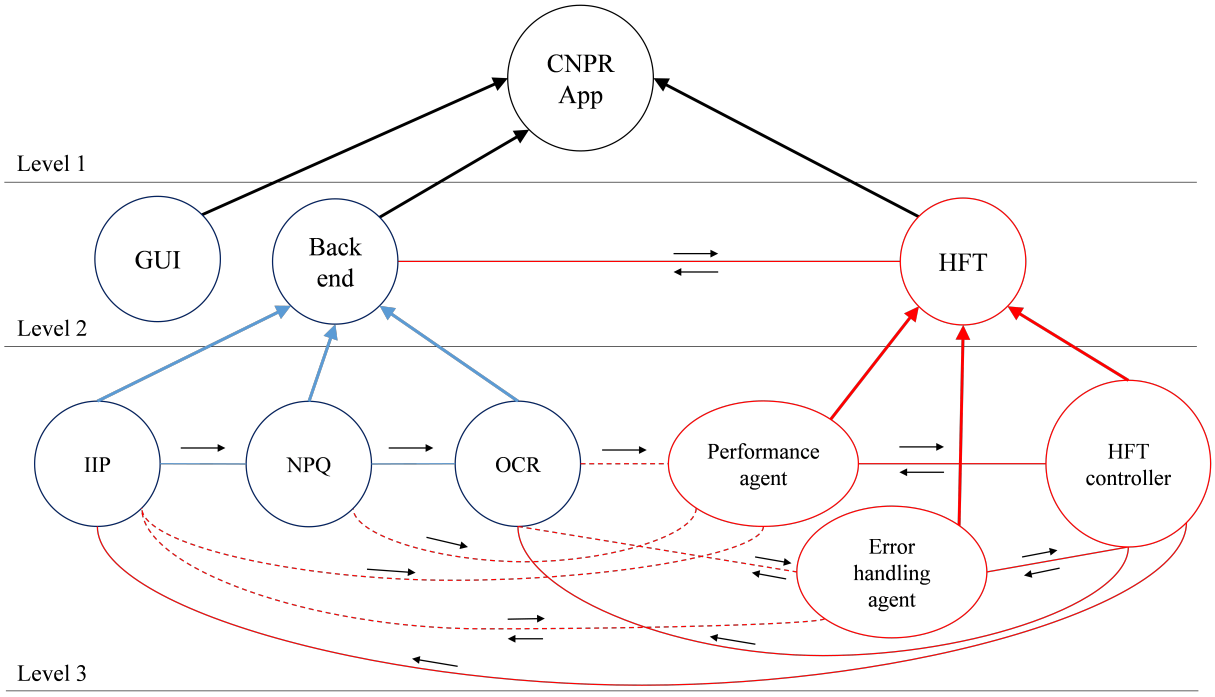

Figure 5.6: Hierarchical model of the system

At Level 1 of the system Order Graph there is only one node "Car Number Plate Recognition Application". Level 2 distinguishes between the HFT part of the system and the functional part, which comprises the GUI and system backend. At Level 3 all crucial components of the system and the HFT part are illustrated. We do not model GUI behaviour, therefore we stop at Level 2 for the GUI. The backend was decomposed to IIP, NPQ and OCR components because it follows the UML structure of the application. The

HFT part is decomposed to the HFT controller, the Performance agent and the Error Handling agent. At Level 4 there is further decomposition to the inner structure of the functional components and the HFT elements. Level 4 is not illustrated here due to the number of elements at this level.

There is unidirectional information flow from the IIP, the NPQ and the OCR components to the Performance agent, because this agent only monitors the components, but it does not affect the control flow of the components. In contrast, the Error Handling agent has bi-directional information flow, since it intervenes in the control flow of IIP and OCR components in order to handle the errors. The interfaces between the agents and components are represented by dashed lines because they are implicit for the components. The HFT controller, in turn, utilises public interfaces of the IIP and OCR components to reconfigure the components and performs fault handling of the application. In addition, there are information flows between the HFT controller and the HFT agents. It can be seen that all control loops mentioned in Section 5.4 exist in this Order Graph, which validates the correctness of the selected HFT architecture.

5.6 Discussion

In this chapter we elaborated the general method for modelling computer-based systems with HFT at the early stages of system engineering. The given method simplifies the modelling process and allows the developer to adjust the system at the early stage to achieve efficient operation after implementation.

As a part of the workflow, the SANs model of the system was built using the Möbius tool. After that the list of interfaces for the HFT control representing the control loops between system components and non-functional properties of the system was obtained. At the same time, a hierarchical model of the system with HFT was created using Order Graphs. The method has been demonstrated with a use case application of UK number plate recognition.

The next chapter is dedicated to the implementation of the HFT architecture for the software application. AOP [45] was applied to implement crosscutting functionality of the application. The structure of the AOP modules is defined by the list of interfaces between the HFT part and the system functional components. These interfaces are defined at the modelling stage.

Chapter 7 is devoted to the evaluation of the HFT architecture. We compare efficiency and maintainability of the software application with the HFT architecture and the software application without the HFT architecture. These applications are functionally identical, however crosscutting concerns (fault tolerance mechanisms, performance monitoring, etc.) are implemented differently. The presented modelling method was applied to model and adjust the application with the HFT architecture in order to prepare it for efficiency evaluation, which involves comparison of performance, resource utilisation and reliability under different operating conditions.

6

Implementation of Holistic Fault Tolerance

Contents

6.1 Introduction

The previous chapters described details of the HFT architecture and HFT modelling. This chapter is devoted to the implementation stage of the HFT engineering process. It describes the principles of software structural quality, analyses the AOP paradigm and its applicability, and finally, presents the guidelines and advice on the implementation of quality software applications with HFT. These guidelines cover critical system components, elements of the HFT architecture, and their interactions. The HFT controller and application components are implemented using the best practices of Object-Oriented Programming (OOP). Implementation of the HFT agents, in turn, leverages AOP to capture crosscutting concerns.

The chapter is organised as follows. The principles that are applied to improve software maintainability, understandability and flexibility are examined in Section 6.2. The AOP programming paradigm is considered in Section 6.3. The reasons for choosing AOP for the implementation of the HFT mechanisms are discussed in Section 6.4. Guidance on the implementation of the application components, the HFT controller, and the HFT agents is presented in Section 6.5. Features of various application domains that could be useful for an implementation of HFT are considered in Section 6.6. Section 6.7 sums up the chapter.

6.2 Principles of software structural quality

The principles that act as guidelines for creation of robust and easily maintainable software systems are described further.

6.2.1 Abstraction

The notion of abstraction is to focus on information that is important in a given context and hide information that is irrelevant [128] in that context. Abstraction is applied to deal with complexity of a computer system. It allows the developer to work with data objects without going into their implementation details. Each object provides a simple interface, while intricate implementation details are encapsulated.

6.2.2 Single Responsibility Principle

According to the Single Responsibility Principle [129, Chapter 8] every module should work on its own task and it should have responsibility over a single part of the software functionality. In addition, in case of modifications, the module should have only one reason of change, otherwise, a split into two or more modules is necessary.

6.2.3 Open/closed principle

The Open/closed principle (OCP) [130, 131] implies that software modules should be open for extension, but closed for modification. This principle ensures that a single change in one module does not cause changes in dependent modules. Moreover, OCP guarantees that system functionality is not corrupted after the extension of the program. Thus, it is more preferable to add new code rather than modify or delete any existing code.

6.2.4 Coupling and cohesion

Coupling and cohesion usually are considered together [132]. The former describes inter-dependencies between modules, while the latter illustrates how the elements of the module are related to each other. Developers are expected to provide high cohesion within each module and loose coupling among modules to reduce complexity, improve readability and support maintainability of the software. Software modules should be easily replaceable in such a way that other parts of the program do not require significant changes after these actions. This task is much easier when the modules are loosely coupled with each other. High cohesion, in turn, means that similar functionality should be placed in one module. This concept is a very good argument why the system FT mechanisms should be placed in a separate module rather than be partially implemented by each individual module.

6.2.5 Modularity

Modularity is a way to design a system consisting of separate modules where each module is responsible for its area of concern [30]. A modular design simplifies the implementation of modules, due to high cohesion inside modules, loose coupling between modules and well-defined interfaces for module interactions. Such an approach creates logical separation between modules, improves maintainability, and allows the developers to implement and test modules independently.

6.2.6 Separation of concerns

Separation of concerns (SoC) [133] is a design principle assuming that a computer program should be divided into distinct features to ensure modularity of the program code. Each of these features or concerns represents a single piece of interest in the program, such as business logic, database access level, user interface or API for external clients. However, some concerns are dispersed across different parts of the program.

These *crosscutting concerns* [36] affect the entire system and cannot be distinguished straightforwardly to fit the OOP style. In object-oriented design these concerns can create a high degree of tangling and affect modularity of the program. This often results in scattering and tangling of the source code responsible for the implementation of crosscutting concerns. Usually it is impossible to decompose crosscutting concerns from the remaining system part during the design and implementation stages of the software engineering process. Typical examples of crosscutting concerns include error and exception handling, caching, monitoring, logging, and security. There are academic and industrial studies [30, 31] referring to FT as a crosscutting concern. For example, error handlers should be reused rather than copy-pasted, whenever it is possible and practical. Hence, when an error affects the whole system but not a single component it should be handled by a designated system-wide action but not by the component itself. Crosscutting concerns were a motivation for the AOP paradigm.

6.2.7 Software reuse

The software reuse [134] principle implies that it is a good practice to use existing functions, patterns and modules in order to reduce redundancy, decrease development time, improve maintainability, and ensure quality of software. This is possible because already created and time-proved software parts are used in the development process. The notion of software reuse infers that the software under construction should be treated in such a way that it will be used for development of new software in the future.

6.3 Aspect-Oriented Programming

AOP [36] is a promising paradigm intended to improve the modularity of computer systems by the separation of crosscutting concerns. It is achieved by extension of the program code behaviour in certain points, without modification of the code itself. AOP languages

provide special mechanisms to specify crosscutting code in separate modules called *aspects*. Each aspect usually contains several advice, which complement or even alter behaviour of the functional code when a certain point of program execution, called *join point*, is reached. Aspects in AOP can be considered as analogues of classes in OOP, whereas advice constructions are analogues of OOP methods. The AOP paradigm assumes that classes should work with functional concerns that can be clearly distinguished from the remaining part of the codebase, whereas aspects group crosscutting concerns that are scattered across the codebase. AOP advice and OOP method are connected when control flow of the program reaches a specified join point.

There are a great variety of AOP extensions for the Java programming language, like AspectJ, Spring AOP, CaesarJ and JBoss AOP. In comparison with other languages, AspectJ is considered a powerful, most mature and most popular language [135, 136] allowing developers to write easily understandable AOP code that supports modular implementation of crosscutting concerns.

AspectJ [45, 136] supports several AOP constructs like aspect, advice and pointcut that rely on the concept of join point. *Join points* are certain points in the program execution workflow, presented by method calls, returns from methods, and exceptions thrown during the method execution. *Pointcut* represents a named collection of join points. There are several kinds of advice in AspectJ. *Before* advice runs before the code that starts at the join point. *After* advice runs after the piece of code that started at the join point. *Around* advice runs instead of the join point. If necessary, the control flow of the method can be altered using an around advice. The next section considers various examples of FT mechanisms implemented with AOP.

6.4 Aspect-Oriented Programming for the implementation of fault tolerance mechanisms

The state-of-the-art studies on applying AOP for the implementation of the system FT are examined in this section.

In [135] the quantitative assessment of exception handling as aspects is provided. The author considers the benefits of using AOP for modularisation of exception detection and exception handling. AOP allows the developer to lexically separate the exception handling code and the normal application code making changes in the AOP code less intrusive and

much simpler.

The paper [137] analyses the claim that AOP facilitates the modularisation of exception handling mechanisms. The authors state that the majority of software development methodologies do not give consideration to the design of a system's exceptional behaviour. It is shown that in some cases AOP could even deteriorate the quality of the system. The main result of the study is that AOP will not improve FT in a system with bad architecture. However, it is able to improve the structure of well designed systems by separating normal and exceptional activities of the system.

Feasibility and evaluation of using AOP for Software Implemented Hardware Fault Tolerance (SIHFT) is presented in [138]. The authors propose to apply AOP in order to avoid tangling of SIHFT code with code related to the main functionality of the program. Fault coverage and performance penalty were used to assess SIHFT based on aspects. According to the experimental results AOP is convenient for programs with SIHFT.

The paper [139] estimates the impact of using AOP and compares AOP with other techniques. The authors measure memory consumption and execution time overhead of the automotive brake controller application after introducing FT mechanisms represented by time redundant execution and control flow checking. These software mechanisms are intended to deal with hardware faults. The implementation is done at a source code level by three approaches: AOP, source code transformation and manual programming in C. Software implemented FT is preferable since it allows the designers to minimise the cost of redundancy by using self-checking and internally fault tolerant electronic control unit (ECU) instead of replicating several ECUs. The authors analysed the pros and cons of AOP for systematic and application specific implementations. At the function level, FT mechanisms have a very high degree of tangling. This is the reason why AOP introduces some performance overheads for systematic implementation. However, when knowledge of the application is leveraged, the overheads of using AOP are similar to those caused by manual programming in C, but AOP is more preferable for the developer since it provides the separation of crosscutting concerns.

Research focusing on facilitating exception handling is presented in [140]. The authors state that the goal of exception handling mechanisms is to distinguish normal code and error handling code. However, when the exception related code is modified, the control flow of the program could be unexpectedly affected. Sometimes it is difficult to locate the place where the exception will be handled or where it was raised. The authors claim

that the main disadvantage of Java-type languages is that there is no link between the exception rising site and the exception handling site. However, the exception control flow is a crosscutting concern and AOP techniques can be applied to facilitate modularity and maintainability in the presence of exceptions. For example, AspectJ [45] provides a way to distinguish normal and error handling code. Thus, AOP can be applied to reduce error likelihood, facilitate software maintainability, improve implementation productivity by providing automated support for developers and make exception control flow more understandable.

Analysis of these studies demonstrated the benefits and feasibility of using AOP for implementation of the system FT mechanisms. In the majority of the examples FT is considered as a crosscutting concern.

6.5 Guidance on the implementation of HFT

This section presents a guidance on the implementation of a software application with HFT where a system-wide FT is considered as a crosscutting concern. Various options can be chosen for the implementation of the HFT mechanisms, such as classes with static members, computational reflection, and AOP. All these approaches have advantages and disadvantages. Classes with static members could be more understandable for the developers who are not experienced in the reflection and AOP principles, but they do not solve the problem with code tangling. Computational reflection [141] can facilitate modularisation of the program and provide the same benefits for software maintainability as AOP. However, reflection works in the run-time and could significantly affect performance of the application. In contrast, the AspectJ [45] compiler weaves an aspect code into a class code at compile time. Thus, a produced Java's bytecode will contain aspect behaviour after compilation. After thorough analysis of these options and examination of existing research (Section 6.4), it was decided to use AspectJ for the implementation of system-wide FT mechanisms. The Java programming language was chosen to demonstrate the HFT implementation since it is a cross-platform, high-level language successfully applied in different application domains.

The HFT architecture assists in following the single responsibility principle. Therefore, functional components are dealing with their direct functional responsibilities, whereas the management of FT, performance and resource utilisation is given to the HFT part of

the application.

The UML class diagram of the application with HFT that is applied as a template to demonstrate the HFT implementation is illustrated in Figure 6.1. Critical functional components and the HFT controller are implemented in Java using pure OOP style. The HFT agents are implemented in Java with the AspectJ [45] AOP extension. The implementation detail of these elements are considered further.

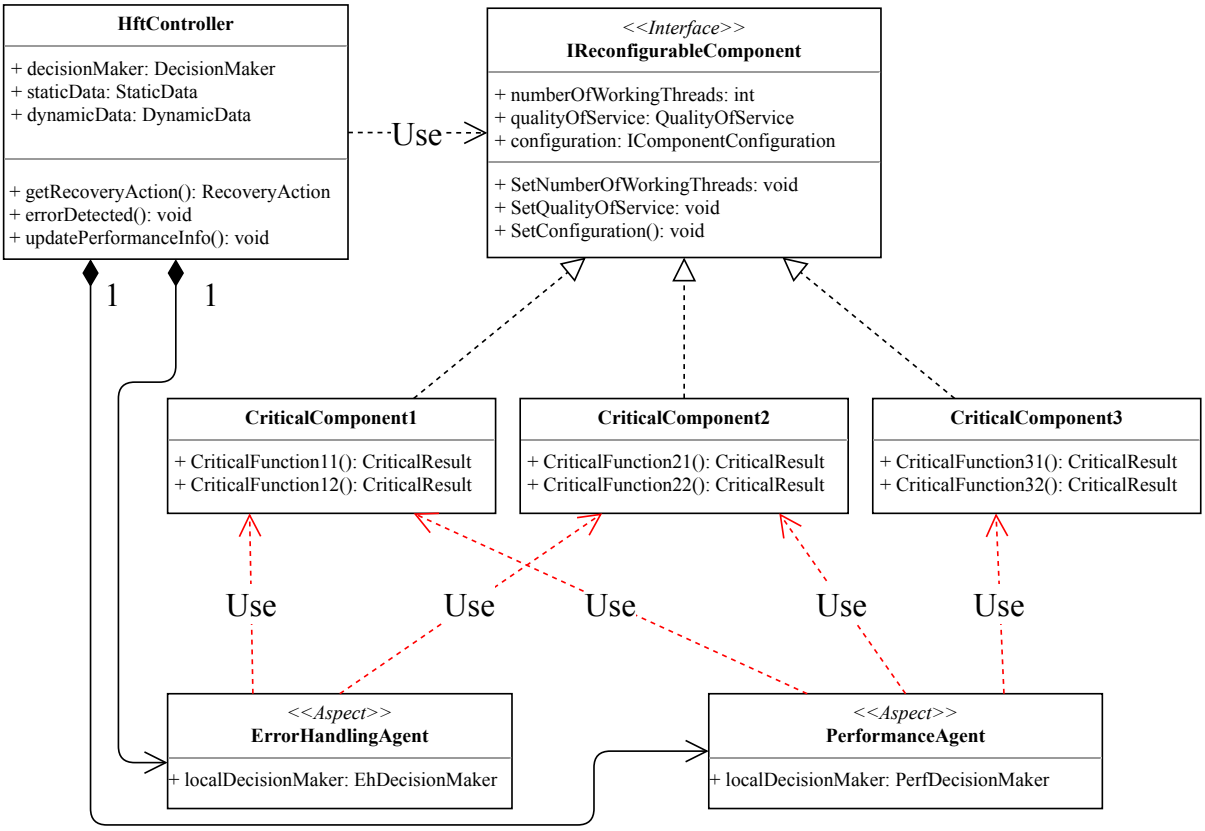

Figure 6.1: The application with HFT

6.5.1 Implementation of critical system components

It is always preferable to implement HFT-ready critical system components. Only in this case would the application leverage all the advantages of the HFT architecture. Otherwise, the application will not use the benefits of fault handling and reconfiguration provided by the HFT architecture.

Critical components should be implemented with redundancy to support HFT mechanisms, however, they should not be aware of the reconfiguration policies, since the latter could change. Instead, critical components should provide an interface for the HFT controller, that will adjust the component's performance, reliability and resource utilisation.

This interface should not reveal the inner structure of the component to avoid overcomplication of the HFT controller.

In addition, the critical components will be implicitly monitored by the HFT agents (these *dependency* relations are depicted in red colour in Figure 6.1). The implicit relationship was chosen to focus on functional behaviour of components. There is no need for critical components to provide a special interface for the HFT agents since the relationship is implicit for the components. However, these components should be developed according to best practices (Section 6.2). Class methods should be meaningful and concise [142]. Only in this case will the HFT agents easily monitor and intervene the control flow of critical components. The preferable structure of the methods in critical component is shown in Figure 6.2. The first method supports several alternates, which is very useful for error handling mechanisms implemented by the Error Handling agent. In addition, both methods are concise, which facilitates the reasoning about their performance and quality of service by the agents.

```
1  public class CriticalComponent1 implements IReconfigurableComponent {
2    // Fields
3     ...
4    // Constructor
5     ...
6    public CriticalResult CriticalFunction11() {
7      switch (m_qualityOfService) {
8      case Low:
9        return LowQualityAlternate();
10     case Medium:
11       return MediumQualityAlternate();
12     case Good:
13       return GoodQualityAlternate();
14     default:
15       return LowQualityAlternate();
16     }
17   }
18
19   public CriticalResult CriticalFunction12() {
20     int [] operationAResult = OperationA();
21     int [] operationBResult = OperationB();
22     return Calculate(operationAResult, operationBResult);
23   }
24    ...
25   // Methods
26 }
```

Figure 6.2: Critical component implementation

6.5.2 Implementation of the HFT controller

There are several ways to implement the HFT controller. Depending on the system scale, the HFT controller and its elements can be implemented using only one class, or several classes (current case), or even separate packages. Since this section considers medium-scale applications, the HFT controller implemented as a class is sufficient.

If the HFT controller is implemented as a standard Java class then the developer should ensure that only one instance of this class is created. This can be resolved by implementing the HFT controller as a singleton class. This is a flexible solution, which guarantees that only one instance of the HFT controller will be available across the application. However, a singleton class, like any other patterns, should be used with precaution to avoid calls to this singleton from all parts of the application. Another option is to implement the HFT controller as a class with a private constructor and only static methods. This implementation has similarities with a singleton class. However, such a class cannot implement the interface and there are no instances of this class. Under such circumstances the HFT agents will need to call the HFT controller explicitly, which increases the coupling between the HFT agents and the HFT controller.

A template for the implementation of the HFT controller is illustrated in Figure 6.3. The HFT controller is implemented as a class and its inner elements (decision maker, static data storage and dynamic data storage) are implemented as separate classes as well.

The Static Data Storage stores all the HFT policies that correspond to the non-functional requirements of the application. The Dynamic Data Storage collects the information about the state of application components. This information is provided by the monitoring mechanisms of the HFT agents. The decision maker has a separate thread for continuous analysis of the states of critical components based on information from the Dynamic Data Storage and the system requirements specified in the Static Data Storage. When it detects that the application could operate faster or more reliably it will reconfigure the application components through their reconfiguration interfaces. In addition, the HFT controller provides several methods that are used by the HFT agents during error handling and performance monitoring activities.

```
1  public class HftController implements IHftController {
2    private StaticDataStorage m_staticData;
3    private DynamicDataStorage m_dynamicData;
4    private DecisionMaker m_decisionMaker;
5    private IReconfigurableComponent[] m_criticalComponents;
6
7    public HftController(IReconfigurableComponent[] criticalComponents) {
8      m_criticalComponents = criticalComponents;
9      m_staticData = new StaticDataStorage();
10     m_dynamicData = new DynamicDataStorage();
11     m_decisionMaker = new DecisionMaker(m_staticData, m_dynamicData,
           m_criticalComponents);
12     PerformanceAgent.setHftController(this);
13     ErrorHandlingAgent.setHftController(this);
14   }
15
16   public RecoveryAction getRecoveryAction(MonitoredFunctions func, Exception exc, int
         attemptNumber) {
17
18     m_dynamicData.errorDetected(func, exc, attemptNumber);
19     return m_decisionMaker.getRecoveryAction(func, exc, attemptNumber);
20   }
21
22   public void errorDetected(MonitoredFunctions func, Exception exc, int attemptNumber) {
23     m_dynamicData.errorDetected(func, exc, attemptNumber);
24   }
25
26   public void updatePerformanceInfo(MonitoredFunctions func, long executionTime) {
27     m_dynamicData.updatePerformanceInfo(func, executionTime);
28   }
29 }
```

Figure 6.3: The HFT controller implementation

6.5.3 Implementation of the HFT agents

Section 6.4 considered the studies that showed the feasibility and benefits of centralisation of the FT management. In the majority of the examples FT is considered as a crosscutting concern and AOP was employed for the implementation of the system FT mechanisms. Thus, the AOP paradigm was chosen for the implementation of the HFT agents. Each HFT agent is implemented as an AspectJ *aspect* and its monitoring and intervention mechanisms are implemented as an AspectJ *around advice* (Section 6.3). Such an approach makes the HFT mechanisms fully implicit for critical application components.

As part of the HFT agents' implementation it is advised to introduce a separate aspect to store all AspectJ pointcuts that are used in an AOP advice of the HFT agents. This would provide a clear access to all join points that are targeted by the HFT agents. In addition, this aspect supports reuse, since the same join points can be used by various agents. An aspect with two pointcuts is shown in Figure 6.4. A pointcut with name

```
1  public aspect Pointcuts {
2    pointcut PointcutCriticalFunction11(CriticalComponent1 component) :
3      target(component) && call(CriticalResult CriticalFunction11());
4
5    pointcut PointcutCriticalFunction12(CriticalComponent1 component) :
6      target(component) && call(CriticalResult CriticalFunction12());
7  }
```

Figure 6.4: AspectJ aspect with pointcuts

`PointcutCriticalFunction11` contains one join point, which will be reached when the method with name `CriticalFunction11` from the class of type `CriticalComponent1` returning an object of type `CriticalResult` is called.

```
1  public aspect PerformanceAgent {
2    private static IHftController m_hftController;
3
4    public static void setHftController(IHftController hftController) {
5      m_hftController = hftController;
6    }
7
8    CriticalResult around(CriticalComponent1 component) throws Exception :
9      Pointcuts.PointcutCriticalFunction11(component) {
10
11     long startTime = System.currentTimeMillis();
12     CriticalResult result = proceed(component);
13     long executionTime = System.currentTimeMillis() - startTime;
14     m_hftController.updatePerformanceInfo(
15       MonitoredFunctions.CriticalFunction11, executionTime);
16     return result;
17   }
18 }
```

Figure 6.5: Aspect implementation of the Performance agent

The HFT agents require an interface from the HFT controller to supply it with information about the system state and to request suitable error handling actions. The usage of this interface by the Performance agent is shown in Figure 6.5. This agent performs monitoring activities only. Firstly, it notes the time before the start of the `CriticalFunction11` method execution. Then, it executes this method (`proceed(component)` instruction). And, finally, it calculates the total execution time and sends this information to the HFT controller.

The Error Handling agent performs monitoring and intervention activities. Figure 6.6 shows the AOP advice that performs the handling of errors occurred in the `CriticalFunction11` method. This advice wraps the call of the original method to the try/catch block. Whenever an exception is raised the advice checks whether the number of attempts

```
1  CriticalResult around(CriticalComponent1 component) throws Exception :
       Pointcuts.PointcutCriticalFunction11(component) {
2
3    int attemptNumber = 0;
4    while (true) {
5      try {
6        attemptNumber++;
7        CriticalResult result = proceed(component);
8        return result;
9      }
10     catch (Exception ex) {
11       if (attemptNumber >= Settings.NumberOfAttempts)
12         return CriticalResult.GetEmptyResult();
13       RecoveryAction ra = m_hftController.getRecoveryAction(
14         MonitoredFunctions.CriticalFunction11, ex, attemptNumber);
15       if (ra == RecoveryAction.Retry)
16         continue;
17       else if (ra == RecoveryAction.Skip)
18         throw ex;
19       else if (ra == RecoveryAction.TryAnotherAlternate)
20         return TryAnotherAlternate();
21     }
22   }
23 }
```

Figure 6.6: Error handling advice with request to the HFT controller

is exceeded and after that requests the most suitable action from the HFT controller. These actions include *retry the operation*, *skip the exception* (allow the method to handle the exception by itself), and *try another alternate* instead of `CriticalFunction11`.

```
1  CriticalResult around(CriticalComponent1 component) throws Exception :
       Pointcuts.PointcutCriticalFunction12(component) {
2
3    int attemptNumber = 0;
4    while (true) {
5      try {
6        attemptNumber++;
7        CriticalResult result = proceed(component);
8        return result;
9      }
10     catch (Exception ex) {
11       if (attemptNumber >= Settings.NumberOfAttempts)
12         return CriticalResult.GetEmptyResult();
13     }
14   }
15 }
```

Figure 6.7: Error handling advice without request to the HFT controller

Figure 6.7 illustrates the AOP advice implementing simple exception handling for `CriticalFunction12` without requests to the HFT controller. Thus, some FT mechanisms can be implemented by the Error Handling agent without participation of the HFT controller.

6.5.4 Discussion of the HFT implementation

The HFT implementation templates presented in this section provide an easy way to analyse and maintain FT-related mechanisms of the application. All system-wide FT mechanisms, performance management and resource distribution techniques can be easily traced through the HFT controller and corresponding HFT agent. If some system-wide modification of the FT-related techniques will be required, then it is very likely that this modification would affect only the HFT part of the application. At the same time, an analysis and modification of critical components would be much easier, since their code will be responsible for their functional tasks only and it will not be tangled with crosscutting code.

It is obvious that implementation of HFT for a particular system would diverge slightly from the presented guidelines, since each application would require its own HFT mechanisms. The contribution of this section is a set of templates and general principles that facilitate implementation of the HFT mechanisms for various applications. The goal of these principles is to ensure that the HFT mechanisms are efficient, understandable and easily maintainable.

6.6 HFT in different application domains

The application domain of the system should always be considered during HFT engineering. Some domains support the features that can be naturally adopted during the implementation of HFT using the guidelines presented in the previous section. This includes leverage of multi-core and many-core systems, and special programming language extensions that will be considered below.

6.6.1 Multi-core and many-core systems

Computer systems with tens, hundreds or thousands of processor cores are called many-core systems [143], whereas multi-core systems have typically only 2-8 cores. Many-core architectures use low performance small cores each of which alone is less productive than a large core, however hundreds or thousands of small cores deliver better performance than ten large cores. Even though we can expect that the throughput will increase with an increasing number of cores, the performance growth is restricted by the percentage of serial

code in the application according to Amdahl's Law [144]. Engineering of efficient many-core systems is now an area of active research focusing on developing scalable methods for structuring complex many-core applications and for efficient parallelisation at the operating system and hardware layers.

Multi-core and many-core systems are likely to become the predominant type of architectures used in the future and it is expected that they will become widely used in safety-critical applications. Borkar proposes [43] that the number and variety of cores will be continually increasing. One of the challenges in the area is that there is a need to understand the trade-off between reliability, performance and resource utilisation (or more specifically energy consumption), since more energy is necessary to support the operation of redundant cores. Another challenge is that FT is typically developed separately for each system layer even though these systems are always built as multilayer architectures.

Ensuring FT of many-core systems is a complex task during the system engineering. The first problem is that when voltage and frequency scaling is applied to reduce power consumption reliability is affected when near-threshold values are used. So there is a need to understand the interplay between energy and reliability. Moreover, modern semiconductors are more vulnerable to faults or negative effects like ageing and variation due to their extra small sizes. Many-core systems can provide redundancy to deal with these problems (e.g. some cores can be used to provide error detection and error recovery for other cores). Our analysis shows that in many-core systems FT is typically applied at the individual layers such as OS, application, communication middleware, memory, etc.

Apart from providing redundancy, some multi-core and many-core systems are heterogeneous [145], which gives the possibility to utilise different computation units for various tasks. This is very useful for reducing power and energy consumption.

Ensuring high performance, low energy consumption, efficient resource utilisation and high reliability, as well as understanding their interplay are the main challenges for all types of many-core systems ranging from large-scale systems, like data centres to small-scale systems, like mobile devices. HFT can tackle these challenges while utilising useful features of multi-core and many-core systems.

6.6.2 Energy Types

The paper [146] describes a new type of system aimed to improve energy-aware programming. It is claimed that application energy management can be improved by *phases* and

modes. The phase is used to illustrate a distinct pattern of program workload and the mode exemplifies an anticipated energy state of the application. The authors assume that a type-based approach is a very wise way to provide energy efficiency of the application depending on the required energy budget.

Energy Types is an object-oriented language for smartphone programming. It has almost the same syntax as Java except additional language features have been developed to support phases and modes directly in the source code. Phases are defined with special phase qualifiers, for example, mathematical calculation phase, input-output phase, graphics phase and so on. Various phases require different CPU utilisation. A similar way is used to work with modes. The difference between phases and modes is that the former are known during compilation time, whereas the latter are chosen during the run-time. In order to adjust CPU energy consumption depending on current phase and mode, the authors use the Dynamic Voltage and Frequency Scaling (DVFS) technique as a low-level energy management.

If the platform where the application with HFT is being developed provides such features as the Energy Types language, then these features should definitely be applied for the implementation of the HFT mechanisms. The HFT implementation should leverage all the possibilities that could improve maintainability and efficiency of the HFT mechanisms.

6.7 Conclusion

This chapter analysed the principles of software structural quality and considered the software engineering approaches facilitating the implementation of FT mechanisms by addressing FT as a crosscutting concern. These ideas motivated an approach for the implementation of HFT. The main contribution of the chapter is a guidance for the implementation of HFT for a medium-scale software application using AOP. The general guidelines for the implementation of the HFT controller, the HFT agents, critical system components, and interfaces between all these elements were provided. These guidelines can be applied to the development of future software applications with HFT.

The next chapter evaluates the AOP-based implementation of HFT with regards to maintainability and efficiency of the FT-related mechanisms. The evaluation was performed by comparing two functionally identical applications, the first is based on the HFT architecture, while the second is implemented according to standard OOP principles.

7

Evaluation of the Holistic Fault Tolerance architecture

Contents

In Chapter 3 an approach for the engineering of HFT was proposed. Engineering steps including a detailed description of the HFT architecture, a modelling method for computer-based systems with the HFT architecture, and implementation of the application with HFT were expanded on in the subsequent chapters. In this chapter, we will talk about evaluation of the HFT architecture implemented with AOP. The evaluation is very important since it demonstrates that the concept of HFT is sound and can be applied in practice for real computer-based systems. There are two main goals of HFT to ensure maintainability of FT mechanisms and achieve an efficient system operation acting upon the interplay between reliability, performance and resource utilisation. In this chapter we verify how HFT addresses both goals and, more precisely, report on the evaluation of HFT efficiency and HFT maintainability.

This chapter is organised as follows. An introduction and the main idea of the chapter are given in Section 7.1. The experimental setup is described in Section 7.2. The HFT maintainability evaluation experiments and the experimental results are discussed in Section 7.3 and Section 7.4 correspondingly. Evaluation of the HFT efficiency is presented in Section 7.5. The role of HFT modelling in the evaluation of the HFT architecture is presented in Section 7.6. Possible overheads of the HFT architecture and ways to deal with these overheads are discussed in Section 7.7. Concluding remarks and a summary of the chapter are given in Section 7.8.

7.1 Introduction

Evaluation of the efficiency and maintainability of a software architecture before deployment is crucial in creating high-quality software. It allows the developers to find out an anticipated quality of the software under development and estimate the difficulties of future maintenance and repairs.

Software maintenance is a crucial phase of the software development life cycle. It is important to facilitate this stage, complying with both functional and non-functional requirements. However, very often the main focus is on the functional features of the application, whereas the FT mechanisms are neglected and as a result do not provide sufficient maintainability and reusability.

Software efficiency ensures that the software operation is as close to optimal operation as is possible and practical. An efficient software application involves minimal waste of

resources, such as time, computational power, and energy.

In the previous chapters it was already presented how to architect, model and implement HFT for a computer-based system. However, to prove that HFT provides benefits in comparison with the standard state-of-the-art approaches, there is a need to evaluate the HFT architecture. It is claimed that the HFT architecture provides efficiency and maintainability benefits, therefore there is a need to evaluate the HFT efficiency and HFT maintainability. The former considers how the system deals with the trade-off between reliability, performance and resource utilisation at run-time, whereas the latter analyses the developer's effort required to accomplish various maintenance works related to crosscutting concerns.

In both cases mainly non-functional requirements of the system are considered because, ideally, the HFT should not affect the functional operation of the system since HFT focuses on FT-related mechanisms and other non-functional requirements. Thus, efficiency evaluation relates to the trade-off between non-functional properties of the system, and maintainability evaluation relates to the developer's experience during maintenance of the system.

In this chapter an experimental evaluation of efficiency and maintainability of a medium-scale software application based on the HFT architecture is provided. This architecture was implemented using AOP, more specifically an AspectJ AOP extension of the Java language [45]. To evaluate the HFT architecture, two versions of the same application were implemented. The first is based on the HFT architecture with AOP, whereas the second was implemented using standard OOP techniques. After that, a set of experiments on both versions of the application was carried out. For the efficiency evaluation the applications were compared by performance, resource utilisation and reliability. During the maintainability evaluation experiments, a number of FT-specific changes were made in the source code of both applications. The experimental results showed that in the majority of cases the HFT architecture is more maintainable with respect to the FT-related modifications and provides better modularity.

The HFT architecture could involve some overheads, such as broken encapsulation or significant dependencies of the HFT agents on functional components of the system. In Section 7.7 these overheads are discussed, analysed from different points of view, and finally, the possibilities of how to deal with and minimise them are considered.

7.2 Experimental setup

The HFT architecture is not suitable for all software applications. It makes sense to implement the HFT part only for the HFT-ready application. Such an application should be able to run in several operation modes, providing various performance and quality of service, and utilising different amounts of resources. In addition, the application components should support reconfiguration and provide the interfaces allowing the external controllers (the HFT controller in our case) to carry out the reconfiguration to optimise the current operation or handle the faults. If the application does not provide any of these features, then it is not reasonable to implement the HFT architecture for it to facilitate application efficiency.

These were the reasons for choosing the case study application for the recognition of UK number plates as an experimental setup. Its components support reconfiguration, while the application is able to run in different operation modes: reliability, performance and fixed time limit. The application has about 6500 lines of code not including third-party libraries. Two versions of the application were prepared to evaluate maintainability and efficiency of the HFT architecture. One version was developed in accordance with the HFT architecture using the guidelines presented in Chapter 6. Before the evaluation, this version was adjusted using the modelling method proposed in Chapter 5, since modelling is intended to ensure that there are no redundant interactions between the functional part of the application and the HFT part. The advantages of this step are considered in Section 7.6. Another version uses a standard approach to implement FT mechanisms. The UML diagrams of both applications are shown in Figure 7.1 and Figure 7.2. The functional part of both versions is the same. The Car Number Plate Recognition component (CNPR) is responsible for the interaction between the user and the application. The user uploads a set of images as an input for recognition. After that the images are sent to the IIP component where these images are processed concurrently. This component undertakes an initial processing of each image and tries to find the number plate area on the image. There are two algorithms for IIP: rectangle detection based on OpenCV and HAAR cascade [125]. If the number plate is found, it is cropped from the image and sent to the Number Plates Queue (NPQ). The OCR component checks the NPQ and, if it is not empty, the OCR component takes the number plate cutout and performs recognition of the number plate. The OCR component has two algorithms as well: Tesseract [126]

and the number plate recognition algorithm described in [127]. The former algorithm recognises the entire string, while the latter algorithm requires to separate the symbols of the number plate string before recognition.

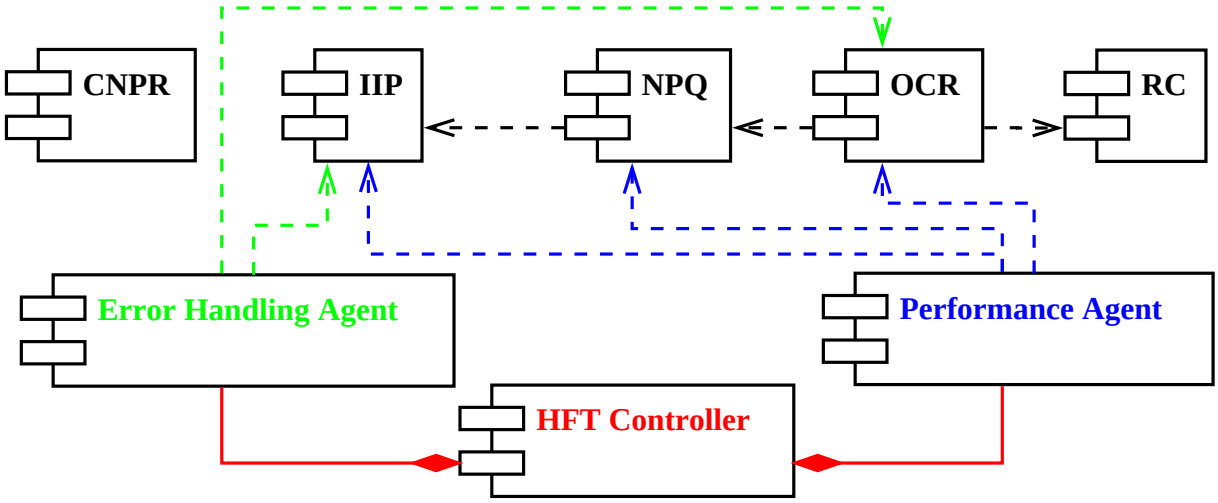

Figure 7.1: The UML diagram of the application HFT version

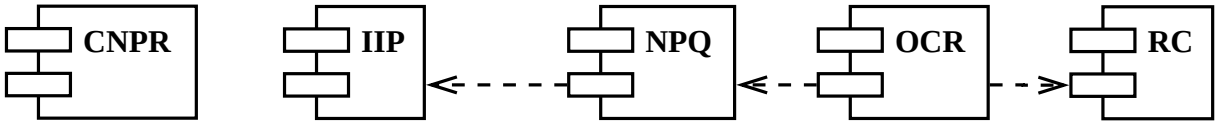

Figure 7.2: The UML diagram of the application non-HFT version

Since third-party libraries with computer vision algorithms were used, it is impossible to be sure that the functions from these libraries will not fail. One problem is inability to detect the required data on the image, which is not an unusual situation for the image recognition operations. Another problem is an exception in the third-party library. Redundant algorithms for the IIP and OCR stages were introduced to perform fault handling and error handling. Depending on the current application state, the error could be skipped or recovered. Recovery includes running another algorithm or re-executing the current algorithm if the fault that caused the error is not permanent. In addition, the application could be reconfigured to use another default algorithm if the current application operation is not efficient. The fault assumptions, error detection and error recovery (Table 7.1) are addressed differently in the two applications. In the HFT version, the HFT controller and the *Error Handling agent* manage FT mechanisms. In the non-HFT version the FT mechanisms are scattered across the functional components and do not have a centralised controller.

Table 7.1: Fault assumptions, error detection and error recovery in the applications

Error	**The HFT detection**	**The HFT recovery**	**The non-HFT detection**	**The non-HFT recovery**
Number plate is not found	The EH agent	The EH agent and the HFT controller	The IIP component	The IIP component
Exception in the library	The EH agent	The EH agent and the HFT controller	The IIP or the OCR component	The IIP or the OCR component
Number plate is not recognised	The EH agent	The EH agent and the HFT controller	The OCR component	The OCR component
Injected CPU exception	The EH agent	The EH agent and the HFT controller	The IIP or the OCR component	The IIP or the OCR component
Recognition fail	The EH agent	The EH agent and the HFT controller	The "Image" object, the IIP and the OCR component	The IIP and the OCR component

To ensure smooth processing of the images, the average NPQ size is kept between 3 and 5 items. If the queue is growing, then the number of working threads used by the OCR component is increased and the number of threads used by the IIP component is decreased. If the queue size is less than 3 and not all images have been processed, then the IIP component creates more working threads, whilst the OCR component reduces its thread pool. In the HFT version performance monitoring is done by the *Performance agent*, and the number of threads for the component is specified through its public interface by the HFT controller. In the non-HFT version, performance monitoring is done by the components themselves and the CNPR component specifies the number of threads for the components.

This specific implementation of the case study is well-suited for the demonstration of the capabilities of the HFT architecture, since the components have spare/redundant algorithms and can be reconfigured. In addition, the application allows trading off of performance and quality of service (or recognition rate). Thus, the HFT controller is able to choose an efficient configuration at run-time. These are the typical characteristics of the applications, for which HFT could be applied.

7.3 Maintainability evaluation experiments

When both versions were ready a number of modifications related to FT functionality were defined and implemented. During the experiments, changes in the functional part of the application were avoided since the main goal of these experiments is to evaluate the maintainability of the FT-related mechanisms in the HFT architecture. Typical examples of the modifications in the FT code caused by changing fault assumptions, deploying of new features, or refactoring the codebase were selected. These were made in both versions of the application and are described below.

7.3.1 Changes in the settings that are used in fault tolerance mechanisms

Sometimes the setting that defines a behaviour of FT mechanisms should be updated. This modification allows us to check whether there is any difference in changing the settings in the HFT and in the non-HFT versions.

7.3.2 Centralisation of thread management for IIP and OCR components

In some software applications, there is no special module for thread management since a chosen framework encapsulates these features and no effort from the developer is required. However, if the application is multithreaded and performs a lot of concurrent operations, it inevitably requires a thread management module. In the HFT version the HFT controller is responsible for crosscutting concerns, so it is the best place to manage and distribute the threads (computational resources) among other application components. In the non-HFT version, a new class was introduced. The main motivation for this modification is the separation of concerns, since dealing with thread allocation and distribution is not a task of the functional component.

7.3.3 Handling of injected CPU error

In the case study a CPU error was introduced to analyse the feasibility of handling hardware errors and exceptions at the software layer with the HFT architecture. This error is not a real CPU error, instead it is being injected with a specified rate into critical functions of the IIP and OCR components. In the given modification, two options regarding

the CPU error were considered. The first option is to introduce the CPU error in both applications and to apply system-wide action for the recovery of the CPU error. Before this modification either version did not have any trace of the CPU error. In the HFT version handling of the CPU error was added to the Error Handling agent with assistance of the HFT controller. In the non-HFT version this error is handled mainly by the component (IIP or OCR) where it was detected, but the component requests current recognition success rate to choose the most suitable action for error recovery. The second option is almost opposite. The handling of the CPU error that was introduced at the previous step is moved to the component (IIP or OCR) where this error was detected. This experiment represents transformation of global error handling to local error handling. Therefore, the CPU error is now hidden from the external modules and will be recovered locally.

7.3.4 Logging diagnostics information

The majority of applications require saving diagnostics information. The logs are used to trace possible problems and errors, and subsequently patch the application. Thus, the current modification consists of adding a special logger to the application. For this modification, the changes were made in such a way that each significant stage of image processing, calls of critical functions and exceptions in critical functions are logged. In the HFT version the Diagnostics agent, which is implemented as an aspect was added. For all functions in the application which should be logged, the corresponding *before* or *after* AOP advice was added. All the modifications are concentrated in a single aspect. For the non-HFT version the class *Logger* with static methods, which is much shorter than the Diagnostics agent, was added. However, all the functions whose calls should be logged had to be modified. Switching off the diagnostics information is implemented approximately equally. However, if there is a need to entirely remove the diagnostics for the HFT version, only one aspect should be deleted or modified, whereas for the non-HFT version a set of functions should be modified by deleting the lines of code with *Logger* class calls.

7.3.5 Reconfiguration logic based on operation mode

Operation mode is a flexible application option intended to provide different quality of service in various conditions. It is logical that error handling may be implemented differently for various operation modes. In reliability mode, it is necessary to apply all available

means to recover the error, while in performance mode the error could be skipped in some cases. The latter option is applicable for errors which will not affect the expected reliability of the system. In the HFT version, the operation modes are managed by the HFT controller. In the non-HFT version, the code related to the operation modes is managed by the designated class. Originally IIP and OCR components were able to work in two operation modes: reliability and performance. Reliability mode means that for the specified performance the system should be as reliable as possible, while performance mode assumes that for the required reliability it is necessary to finish all tasks as fast as possible.

7.3.6 "Complex" error detection

In many cases error detection is not a trivial task. Very often the developer needs to check several components or analyse the result of several functions in order to detect an error. This modification involves detection of the error by checking the result of two functions. Low quality of the result in the IIP component and consequent low probability of successful recognition in the OCR component are considered an error. It should be noted that these two conditions separately are not supposed as errors. Error recovery involves reprocessing of the image with "better" settings at the IIP component. In the HFT version the Error Handling agent has the trace of the processing for each image. Thus, the detection of the given situation can be added at the OCR stage. The error recovery action is requested from the HFT controller. For the non-HFT version the information about image processing steps was added to the object that stores the image.

7.4 Maintainability evaluation

In this section the goal of the experiments that measured a set of realistic modifications in the FT-related code is explained, the metrics of the source code maintainability are considered, and the results of the conducted experiments are discussed. These experiments show advantages and limitations of the HFT architecture with respect to the modification of crosscutting concerns related to FT.

7.4.1 Goal of the experiments

In many cases FT is addressed as a crosscutting concern of the application which is why it should be separated from functional modules to improve modularity. It is claimed

that HFT can be beneficial for small- and medium-scale software applications, especially for those which are adjustable based on an interplay between reliability, performance and resource utilisation during the run-time. However, the counter-argument is that HFT would make it harder to maintain the application. The problem could arise due to an implicit coupling between the HFT agents and application components, significant dependence of the HFT controller on the HFT agents and application components, and global knowledge of the HFT controller about the application. In addition, the HFT agents could amend the control flow of the monitored functions, which is not always considered as a benefit for the maintainability of the application.

The aim of this part of the work is to gain empirical knowledge of the positive and negative effects of the HFT architecture on software maintainability. To show the feasibility of applying the HFT architecture experiments evaluating the HFT architecture were carried out. During these experiments, a longitudinal study was performed and the effects of the HFT on software maintainability were analysed.

The evaluation is based on comparison of the efforts required for the implementation of the modifications in two versions of the same application. The first version is implemented in the OOP style with crosscutting functionality implemented in AOP, whereas the second version uses only a pure object-oriented approach. Although the implementations are different, these applications are functionally identical. For simplicity, the application with AOP is called the HFT version and the OOP-based application as the non-HFT version.

This type of evaluation has been chosen since OOP is very wide-spread in modern software development. Thus, it was decided to compare the maintainability of the "standard" solution implemented in OOP-style and the proposed solution that is still OOP-based, but its HFT functionality implemented with AOP. The experiments were designed to understand the challenges that could be faced by developers during the maintenance of FT functionality in the HFT architecture. In addition, we can reason about the complexity of the HFT maintenance in comparison with the popular OOP approach. The modifications chosen for the experiments represent typical changes and bug fixes in the FT mechanisms of a medium-scale software application during maintenance. The evaluation should show how significantly the source code of both applications is affected by each modification. In addition, it should show how easy it is to find the place (class/aspect and function/advice) where the modification should be done. It should be noted that more changes in the

source code, especially related to modifications or deletions of the code, could introduce new bugs. This is the reason why such modifications are considered as non-preferable in comparison with adding new code. However, less new code in a centralised place means less effort required for maintenance. Thus, the experiments will show the differences between two versions and reveal advantages and disadvantages of the HFT architecture for maintainability of FT techniques.

7.4.2 Metrics of the source code maintainability evaluation

The existing approaches to evaluating the reusability, modularity and changeability of program code are considered in this section.

A number of metrics for object-oriented design are proposed in [147]. These metrics illustrate the complexity of class methods (Weighted Methods per Class), coupling between classes in the package (Coupling between object classes) and cohesion illustrating cohesion of the classes (Lack of Cohesion in Methods, which implies that classes should be divided into subclasses to reduce complexity of the original class).

Metrics of AOP code are considered in [148]. These metrics comprise OOP metrics and AOP-specific metrics, such as number of modules affected by the given aspect (Crosscutting Degree of an Aspect), and number of aspects whose advice could be triggered by operations in a given module (Coupling on Advice Execution).

Due to the scope of the application, the metrics like Weighted Methods per Class, Depth of Inheritance Tree, Number of Children, Crosscutting Degree of an Aspect or Coupling on Intercepted Modules were not applied, since these metrics are more suitable for complex systems. The *maintainability index* (MI) metric for assessing software maintainability was proposed in [149]. It is calculated as a factored formula. However, MI is criticised in [150] for the lack of understandability and difficulty of root-cause analysis.

The *Lines of Code* metric is easier for calculation and understanding and suitable for estimation of the developer's effort. The Lines of Code metric was successfully applied to the evaluation of exception detection and handling with AOP in [29] where the main outcomes of the study are mainly based on comparison of the lines of code.

The following metrics of maintainability evaluation were chosen: *lines of code affected*, *functions affected* and *classes affected*. These metrics represent the volume of code affected by the modifications. In addition, each metric is subdivided into three parts. The first is

Quantity of Added (lines of code, functions, classes). This metric is less critical and more preferable since a new feature was added only without changing any existing functionality. The second metric is *Quantity of Modified*. In this case, these is a need to check all places where the modified code (e.g. functions) is used or called. The third metric is *Quantity of Deleted*. If there are deleted functions, it is inevitable that some parts of the program require modification in the places where deleted functions were called.

7.4.3 Evaluation results

Table 7.2: The HFT version

Changes	**Lines of code**			**Functions / Advice**			**Classes / Aspects**		
	A	**M**	**D**	**A**	**M**	**D**	**A**	**M**	**D**
Settings	0	N	0	0	0	0	0	1	0
Thread management	32	17	5	7	12	5	0	7	0
CPU exception (holistic handling)	7	9	0	0	10	0	0	3	0
CPU exception (local handling)	16	2	5	0	3	0	0	3	0
Diagnostics info	106	0	0	17	0	0	1	1	0
Reconfiguration logic based on OM	24	2	0	1	2	0	1	1	0
“Holistic” error detection	47	22	3	3	8	0	0	3	0

Table 7.3: The non-HFT version

Changes	**Lines of code**			**Functions / Advice**			**Classes / Aspects**		
	A	**M**	**D**	**A**	**M**	**D**	**A**	**M**	**D**
Settings	0	N	0	0	0	0	0	1	0
Thread management	38	15	5	8	12	5	1	6	0
CPU exception (holistic handling)	32	9	0	2	10	0	0	3	0
CPU exception (local handling)	0	2	10	0	3	2	0	3	0
Diagnostics info	60	0	0	2	13	0	1	6	0
Reconfiguration logic based on OM	75	0	0	4	2	0	2	1	0
“Holistic” error detection	63	18	3	5	9	0	1	3	0

According to the open/closed principle [130, 131] it is better to add new code rather than modify or delete the existing code. Thus, if some segment of code (e.g. new line, function

or class) was added, it is more preferable than modification or deletion of the existing code. This is how the modification is evaluated in the experiments. The experimental data is presented in Table 7.2 (the HFT version) and Table 7.3 (the non-HFT version). *A*, *M*, *D* column headers mean *added*, *modified* and *deleted* metrics correspondingly.

The first modification relating to the *changes in the settings* was the simplest and showed predictable outcomes. In both cases, it was necessary to modify only one file and for each setting only one line of code was modified. Thus, the change of N settings requires the change of N lines of code. This modification alone cannot be used for reasoning about the HFT architecture.

The *thread management modification* metric did not show significant differences between the two versions. This modification was motivated by the separation of concerns. In addition, thread management could affect performance of the application. We attempted to separate image processing activities and thread management activities in the functional components. The reason that there is no difference between the two versions is that the code managing the threads was already well structured. After modification this code was placed in a designated module in both versions. AOP was not directly applied for this task in the HFT version. The only use of AOP in this modification relates to performance monitoring in the Performance agent.

It is not a trivial question where the error should be handled. Many approaches propose to recover the error in the place where it was detected. However, the component that detected the error is not always aware how this error would affect the application. This leads to the situation when error recovery is implemented to deal with a worst-case scenario, or sometimes the error is not taken into account. At the system component, there is not enough information about the best option for error recovery. The choice significantly depends on the rate of this error. If the error rate is stable and does not have big deviations from the average value, it would be more convenient to recover the error where it was detected. However, when the error rate is not constant and the fault causing the error is intermittent then it would be more convenient to recover the error holistically taking into account the entire system state. Whenever the developer had more information about the error, the recovery would be much more efficient. If the error will not significantly affect the system operation, it could be skipped. Such a scenario is acceptable for the systems that process large amounts of data and there is an allowance for a certain rate of failed operations.

The two following experiments were used to evaluate the efforts required for the implementation of different approaches of CPU error handling. The first approach is *handling of the CPU error by holistic action*. The HFT version was much better for this modification. Only 7 Lines of Code (LoCs) were added in the HFT version, whereas the non-HFT version required 32 LoCs. Moreover, two functions were added to the non-HFT version. A reason for success behind the HFT version is a centralised mechanism for handling various errors. Error handling is performed by the Error Handling agent, which requests suitable recovery action at the HFT controller. So, the information about the CPU error was just added to this centralised handler. In the non-HFT version a CPU error handler was implemented in all places it could be raised.

The second approach is *local handling of the CPU error*. Thus, the CPU error handling was moved to the component where this error was detected. External classes will not be aware about this exception. The metrics for both application versions are not very different. The HFT version required to add 16 LoCs, whereas no new code was added to the non-HFT version. However, only 5 LoCs were deleted in the HFT version, while in the non-HFT version 10 LoCs were deleted. In addition, 2 functions were deleted in the non-HFT version. All other metrics are the same. Hiding (or suppressing) of CPU exception inside the class where it was detected does not demonstrate the benefits of the HFT architecture because the goal of HFT is the opposite. The problem for the HFT architecture here is that the CPU exception was handled by the Error Handling agent and by the HFT controller. The change required to delete all this code and handle the exception inside the IIP and OCR classes. We will get the benefits if the class is allowed to propagate this exception and then handle it with the HFT controller. This was shown in the first approach of CPU error handling.

Logging and grouping the diagnostics information is an important part of a computer-based system especially at the initial stages of system exploitation. However, the source code responsible for saving and processing of the diagnostics information does not contribute to system functionality. Moreover, if this code is tangled with functional code it is difficult for the developers to search the bugs and add new features to the system. Thus, there is a need for textual separation of the functional and diagnostics code. In addition, there should be a possibility to switch off the diagnostics if the problem is resolved or in order to provide better performance during high system load. AOP provides such features. The developer can specify which information should be logged without

modification of the functional code. In the OOP approach, it is necessary to add calls to a special object whenever this information should be logged. In our experiments related to logging the diagnostics information the HFT version was better by a majority of the metrics. It loses by lines added and functions/aspects added metrics, 106 against 60, and 17 against 2 correspondingly. However, the HFT version is simpler and more intuitively understandable, since all changes are made within one file, while in the non-HFT version 6 files and 13 functions had to be modified. The code in the non-HFT version became less readable. Even though the HFT version requires more lines of code, it was necessary to add only 2 functions and 15 AspectJ advice. No modification of the functions or lines of code is required and only one aspect was added. If exception logging only is considered, then the non-HFT version will lose by the "lines of code" metric as well.

Add reconfiguration logic change. The HFT version already had some functions that were reused for this change. The non-HFT version required a new class with new functions. Almost all metrics are better for the HFT version. Moreover, it requires 3 times fewer lines of code.

Complex (or holistic) error detection is a very typical modification for modern software. System requirements are constantly clarified and it is logical that in some cases certain errors could be detected not only by one component, but by monitoring of two or more system components. Even though each state of the separate components is not considered as an error, the combined states of the components are the error. This experiment clearly illustrated the advantages of the HFT architecture. The HFT version requires fewer new LoCs, slightly more modifications in LoCs, and fewer new and modified functions.

Regarding the combined analysis of lines of code affected, functions/advice affected and classes/aspects affected metrics, modifications related to holistic error handling, introducing reconfiguration logic and diagnostics clearly showed the advantages of the HFT architecture. These modifications are very likely FT-related modifications of the application and the HFT version was better for these modifications. The HFT architecture will not give the benefits for handling of local errors that are related to the inner operation of the application components. However, even in this case the HFT architecture based on AOP would not be worse than the standard OOP approach.

7.4.4 Modularity and maintainability of each version

Some metrics such as cohesion, coupling, separation of concerns and changeability do not directly depend on affected LoCs or functions. Changeability of the FT mechanisms was mainly better in the HFT version. The HFT version provides better cohesion with regards to performance and error handling code. In the HFT version this code is not tangled with functional code. Thus, it provides better cohesion in functional components and in the HFT part. In the non-HFT version, performance monitoring and error handling code is significantly tangled with functional code, which increases coupling and decreases cohesion.

The HFT version provides clear separation of performance management, resource utilisation management, FT management and operation mode management. In the non-HFT version the separation of crosscutting concerns is almost not supported. Due to the scope of the case study, it was convenient to use one module (the HFT controller) to manage all these concerns. For larger applications, it would make sense to develop dedicated controllers for each of these concerns and coordinate the functionality of these concern-specific HFT controllers.

The HFT version introduces an implicit coupling between the HFT agents and the monitored application components. Certain modifications of the inner structure of the components would require modification of the HFT agent. However, this is the cost of better cohesion and separation of crosscutting concerns.

7.5 Efficiency evaluation

In this section evaluation of HFT efficiency is discussed. The two versions of the case study application are compared by performance, reliability and resource utilisation. In the HFT version, the HFT controller is responsible for adjusting the system components to provide an efficient operation. In the non-HFT version, the functional components are more complex and they are self-adjusted, since there is no centralised control. If there were a centralised control in the non-HFT version, this would be another implementation of HFT. This is the primary reason for choosing the given evaluation approach. Intuitively, the non-HFT version will be better at optimising the individual components, since each crucial component is itself responsible for the optimisation. However, without the evaluation, it is very difficult to say how the efficiency of the entire application will be affected.

In the evaluation, three crosscutting parameters are monitored and adjusted:

- Performance (average time for image processing).
- Reliability – percentage of successfully recognised number plates, or recognition rate.
- Resource utilisation. In the given case study resource utilisation is the number of CPU threads allocated among application components.

Resource utilisation can be considered differently for different types of devices. On the personal computer or the laptop, which were used for the experiments, energy consumption is not a crucial factor. In addition, due to the specifics of the application we do not experience a lack of disk space or RAM. Therefore, we focus on the CPU usage only. For other types of application, the resource utilisation can be expressed by RAM, disk or network usage. In addition, if the application can be used on a device which has only an autonomous power supply, it is reasonable to analyse power and/or energy usage as a resource utilisation.

7.5.1 Interplay between reliability, performance and resource utilisation

In the evaluation, three non-functional parameters: reliability, performance and resource utilisation are considered. The better application, in current conditions, shows a better parameter value when two other parameters are fixed. For example, the application that runs faster, will be considered better, when reliability and resource usage properties of both applications are the same.

The extreme cases—where one or two parameters are fixed with their lowest possible values—were avoided. If the system is allowed to be totally unreliable, then the best strategy for both applications would be just to fail fast to ensure the best possible performance. Such extreme cases can say neither about the advantages nor the disadvantages of HFT. This is the reason why we focus on the test cases without a trivial solution.

7.5.2 Dynamic adjusting of the applications

Both experimental applications should constantly monitor and, if necessary, adjust their performance, reliability and resource utilisation. The application with the HFT architecture is being adjusted by the HFT controller. When the HFT controller detects that the

application could operate faster with the same reliability and same resource utilisation, it reconfigures the application through the public interfaces of the system components. The non-HFT version, in turn, is being adjusted at the component level, i.e. each crucial component is responsible for its own optimisation.

7.5.3 Running the experiments

In the efficiency evaluation experiment we analyse how the applications operate at run-time by processing the same input data. 100 .jpg images with UK number plates (on the cars) were used for the experiment. It does not make sense to upload the images without number plates to check how the recognition algorithms would deal with that because even with the number plates on the images, the algorithms cannot provide a 100% recognition rate.

For the efficiency evaluation, the total processing time of a set of images, recognition rate of image processing, and a number of working threads allocated is checked.

To optimise the resource usage, the HFT controller can specify how the available CPU threads will be distributed among the IIP and OCR components. In the non-HFT version, the components are doing this task by themselves, based on the NPQ size.

It is preferred when the system operates as reliable as possible. However, even though this is achievable, it is not always reasonable to provide such quality of service. On top of requiring many more resources it could slow down the application and deteriorate the availability – readiness to provide the correct service. Therefore, sometimes it is reasonable to reduce reliability requirements in order to provide better performance and distribute the resources more efficiently.

To provide the flexibility for the trade-off search between these parameters the computer system should be adjustable. In our case study, there are several possibilities for the adjustment. The IIP component has two diverse algorithms for the initial image processing and searching of the number plate: rectangle detection and HAAR cascade. Each algorithm has advantages and disadvantages in certain conditions. Moreover, their behaviour depends on the input images.

The input images are classified into three categories: small (less than 200 Kbyte), medium (200 Kbyte–1 Mbyte) and large (1 Mbyte–7 MByte). Approximate processing times are given in Table 7.4 and Table 7.5. It should be noted that even if the size of the images is

equal, the processing time (in one application version) could differ significantly due to the image data itself. For instance, the rectangle detection algorithm of the IIP component would require much more time if the image was taken in front of a brick wall, a building with windows or some other structure with a rectangular pattern.

Table 7.4: Execution time in performance mode

Run	The HFT version (ms)	The non-HFT version (ms)
1	881	1029
2	947	995
3	910	1053

Table 7.5: Execution time in reliability mode

Run	The HFT version (ms)	The non-HFT version (ms)
1	1317	1531
2	1342	1448
3	1330	1420

Table 7.6: Recognition rate with fixed time limit

Run	The HFT version	The non-HFT version
1	75%	70%
2	72%	71%
3	78%	68%

7.5.4 Efficiency evaluation results

The experiments showed that the HFT version is faster than the non-HFT version in the performance and reliability modes. In the performance mode, the HFT version required about 880–950 ms to process the set of images, whereas the non-HFT version required about 990–1050 ms. In the reliability mode the HFT version was faster as well, 1310–1340 ms against 1420–1530 ms. Moreover, when the execution time is fixed (1250 ms), the HFT version provides a better success rate (Table 7.6).

7.6 Role of modelling in the design and implementation of the case study

The HFT modelling (Chapter 5) assisted in defining the control loops between the HFT elements and functional system components. The HFT version of the application was designed and implemented in an efficient manner, without unnecessary interactions between the functional components and the HFT part. This allowed the HFT version to outperform the non-HFT version, while providing better maintainability. Thus, HFT modelling is an important step in the engineering of computer-based systems with the HFT architecture.

7.7 Overheads of the HFT architecture

In the proposed architecture, there are two types of interaction between the HFT controller and application components. The former link is asynchronous, which uses public interfaces of the application components. When the HFT controller starts reconfiguration of the application, it uses these interfaces to make necessary adjustments in the application components. Since the link is asynchronous, there is no high risk of locks or bottlenecks. However, the latter link is synchronous and it is implicit for the application component. This interaction is initiated by the intervention logic of the HFT agent, when the agent requests the HFT controller for a suitable action. Here is a risk of dead-lock and delays for performance-intensive applications. These applications require a special attention to the implementation of the synchronous part of the HFT controller in order to avoid dead-lock and bottlenecks. It is advised to apply HFT modelling (Chapter 5) to define which interactions between the HFT elements and the application components are required and which interactions could be omitted since they will not provide benefits for the application efficiency.

7.8 Results and discussion of the HFT evaluation

During their lifetime software systems require various maintenance work to add new features and fix discovered bugs. These modifications are related to functional and non-functional features. In this chapter, an experimental evaluation of the Holistic Fault Tolerance architecture based on the typical changes of FT-related code was presented.

We started by justifying the choice of modifications and evaluation techniques. Then the aspect-oriented implementation proposed for HFT was evaluated by conducting its experimental comparison with a standard object-oriented FT implementation. According to the experimental results there are clear benefits of using the HFT architecture implemented with AOP. In most cases, the HFT version is more efficient and it is better for maintainability. Thus, the HFT architecture can be applied to improve maintainability of FT mechanisms and ensure higher modularity of the source code in the application.

The maintainability evaluation showed that for the set of typical changes in FT-related code used to evaluate maintainability, the HFT version (AOP approach) proved to be easier to maintain as opposed to the non-HFT version (pure OOP approach) in most cases. In the remaining cases, an almost identical effort was required to implement modifications in the HFT and non-HFT versions. However, there was no such modification for which the HFT version would be worse than the non-HFT version. In addition the HFT version provides much better cohesion and separation of concerns than the non-HFT version.

The efficiency evaluation confirmed that the HFT version of the case study is better performing as compared to the non-HFT version with regard to the trade-off between reliability, performance and resource utilisation.

In the next chapter the research outcomes will be discussed, the limitations of HFT will be considered, and research directions for future work will be given.

8

Discussion and Conclusion

Contents

8.1 Discussion

Engineering fault tolerance for a computer-based system is not a trivial task. It is impossible to find a universal solution that would fit all occasions. However, it is feasible to provide certain patterns, techniques, and guidances that would improve maintainability, simplify the understanding and ensure the efficiency of fault tolerance mechanisms for computer-based systems. This thesis proposed a methodology for designing computer-based systems with Holistic Fault Tolerance. The main motivations for introducing the concept of Holistic Fault Tolerance are related to the growing problem with efficiency and maintainability of fault tolerance mechanisms in modern computer-based systems.

The thesis started with describing a research area and inspirations for the idea of Holistic Fault Tolerance. The importance of systematic and rigorous engineering of fault tolerance mechanisms to ensure dependability of computer-based systems, and the problems of fault tolerance complexity, maintainability, and misuse were discussed in Chapter 1. The existing approaches to fault tolerance engineering and system structuring that tackle these problems and a critical analysis of their limitations was presented in Chapter 2. After that in Chapter 3 an engineering concept for systems with Holistic Fault Tolerance was formulated and the engineering stages required to develop computer-based systems with HFT were described. Holistic Fault Tolerance solves the problems stated in Chapter 1 by considering system fault tolerance as a crosscutting concern and provides a holistic view of system-wide fault tolerance mechanisms simplifying their understanding and improving their efficiency. The three steps for engineering Holistic Fault Tolerance were presented in Chapter 4, Chapter 5 and Chapter 6. An architectural blueprint describing how to structure computer-based systems with Holistic Fault Tolerance was provided in Chapter 4. The modelling method for such systems was presented in Chapter 5. The patterns and guidelines to implement the HFT architecture using AOP were given in Chapter 6. Maintainability and efficiency of computer-based systems with the HFT architecture were evaluated in Chapter 7.

8.2 Contributions of the study

The thesis focuses on the concept of Holistic Fault Tolerance for computer-based systems. The idea itself and engineering steps including design, architecture, modelling, and implementation that are required to implement a computer-based system with Holistic Fault

Tolerance were covered.

The main contributions of the research are listed below:

- The concept of Holistic Fault Tolerance and all relevant stages required to engineer a computer-based system with Holistic Fault Tolerance are presented.
- Basic elements of the Holistic Fault Tolerance architecture with their interfaces and interactions are described in detail.
- A modelling method intended to facilitate the design and implementation of computer-based systems with the Holistic Fault Tolerance architecture is explained.
- The guidelines on how to implement the application with Holistic Fault Tolerance in the Java programming language using AspectJ AOP extension are provided.

In Subsection 3.2.2 the challenges of FT engineering, such as tolerating new and residual faults, dealing with unanticipated failures, handling more than one concurrent fault activation, and ensuring system protection after modifications were stated. Holistic Fault Tolerance addresses all these challenges by applying a systematic approach to system-wide FT mechanisms during all stages of the development life cycle. Special FT techniques can be implemented by the HFT controller and Error Handling agent to tolerate new and residual faults and to handle two and more concurrent faults. This would minimally affect the functional part of the system, while the mechanisms responsible for tolerating these faults will be implemented in a modular way simplifying deployment, testing and future modifications. With regards to concurrent fault activation, HFT provides the possibility of centralised monitoring of critical components in order to detect and recover the errors caused by concurrent fault activations in different parts of the system.

An example of an unanticipated failure is an unhandled exception in the software application. Some programming languages provide the means to catch all unhandled exceptions in one place. However, it is not advised because the system state could be corrupted because of unhandled exception. The better option could be to let the application crash and use a special watchdog that will restart the application after failure. This watchdog should be dependable itself. In addition, it is necessary to ensure that the system will be able to start after the unhandled exception and there are no inconsistencies with data that were saved to the disk (these corrupted data may prevent the application from restart). The HFT controller can implement the required watchdog possibilities.

To ensure the system protection is verified after modifications and repairs FT should be addressed from the earliest stages of the system development life cycle. This approach is in line with the concept of HFT presented in the thesis. Moreover, HFT maintainability evaluation experiments were conducted to ascertain that system functionality is not corrupted after maintenance work on HFT mechanisms.

The proposed concept was evaluated with regards to efficiency and maintainability of fault tolerance mechanisms. It was shown that the application version implemented according to the Holistic Fault Tolerance architecture using AOP programming illustrates better efficiency of fault tolerance and fault handling mechanisms than the system implemented using a standard OOP approach. In addition, the Holistic Fault Tolerance version provided better maintainability of fault tolerance mechanisms due to the separation of crosscutting concerns.

8.3 Applying HFT to computer-based systems

As was shown, Holistic Fault Tolerance can be applied to computer-based systems to improve modularity and assist in the separation of crosscutting concerns taking into account efficiency and maintainability of computer-based systems. It is more convenient to implement the system with Holistic Fault Tolerance from scratch, since it is easier when functional system components are HFT-ready from the initial steps of system engineering. However, there is still a possibility to introduce Holistic Fault Tolerance into existing systems. The system components should be modified and updated accordingly, by providing all necessary interfaces to the elements of the Holistic Fault Tolerance architecture. In addition, system-wide fault tolerance mechanisms should be redesigned to be managed by the Holistic Fault Tolerance part of the system.

Due to the limited resources available this thesis does not cover all possible applicabilities of Holistic Fault Tolerance. Examples were given for small- and medium-scale software applications. However, Holistic Fault Tolerance is suitable for various systems and it can encompass more layers and components of computer-based systems including hardware.

8.4 Limitations of Holistic Fault Tolerance

The presented research concentrated on the trade-off between maintainability, reliability, performance and resource utilisation of computer-based systems. These are very impor-

tant non-functional characteristics of the system. However, such factors as cost and time of developers' training or willingness of paradigm change by the companies were not considered. Many industrial bodies are quite reluctant to introduce new approaches especially if these approaches do not involve an immediate financial profit. Performance issues are often resolved by upgrading the hardware, while resource usage (sometimes even waste) is very often ignored. Thus, the concept of Holistic Fault Tolerance does not address organisational issues yet.

In addition, system security was not considered in the context of Holistic Fault Tolerance. Security along with fault tolerance and performance is a crosscutting concern and it could be managed by elements of the Holistic Fault Tolerance architecture. However, it is out of scope of this research.

Another limitation of the Holistic Fault Tolerance architecture is that its elements (the HFT controller and the HFT agents) are not adaptable themselves, even though they are used to adjust and configure the functional part of the system. The configuration and adjustment of the HFT elements is performed during development and apart from working under different operation modes there is no significant adjustment of the HFT controller and the HFT agents. In some cases this could prevent the system being as efficient as possible because of redundant activities of the HFT part.

8.5 Future work

As a future work there is a plan to elaborate the idea of *Adaptive Holistic Fault Tolerance* where the elements of the Holistic Fault Tolerance architecture will be self-tunable. This self-tuning will be achieved by introducing or switching-off the HFT agents and reconfiguring the HFT controller at run-time depending on the chosen operation mode, system loading and system state. This will help to ensure the efficiency and scalability of the Holistic Fault Tolerance part of the system.

8.5.1 Adaptive Holistic Fault Tolerance

As mentioned in Section 8.4, the proposed Holistic Fault Tolerance architecture has some limitations regarding the participation of the Holistic Fault Tolerance part in adjusting an entire computer-based system. It seems logical that depending on the system state, the whole system, including the Holistic Fault Tolerance part, should be adaptable and

reconfigurable, but not only the functional part. For example, some HFT agents can be switched-off and other agents can be reconfigured depending on the system state. The HFT controller could select different sets of functional component activities to be monitored under different system loading. This is an area of further research.

8.5.2 Universal approach to applying HFT

The approach presented in the thesis was examined on small- and medium-scale software applications. However, various cloud and distributed systems are common nowadays and there is a need to ensure that Holistic Fault Tolerance is suitable for such systems and these systems can benefit from Holistic Fault Tolerance. Future work will include an in-depth modelling and thorough analysis of all implications of Holistic Fault Tolerance on large-scale systems.

Bibliography

[1] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, pp. 11–33, Jan. 2004.

[2] J. B. Goodenough, "Exception Handling: Issues and a Proposed Notation," *Commun. ACM*, vol. 18, pp. 683–696, Dec. 1975.

[3] R. H. Campbell and B. Randell, "Error recovery in asynchronous systems," *IEEE Transactions on Software Engineering*, vol. SE-12, pp. 811–826, Aug 1986.

[4] F. Cristian, "Exception Handling and Software Fault Tolerance," *IEEE Transactions on Computers*, vol. C-31, pp. 531–540, June 1982.

[5] M. Kaâniche, J.-C. Laprie, and J.-P. Blanquart, "A framework for dependability engineering of critical computing systems," *Safety Science*, vol. 40, no. 9, pp. 731–752, 2002.

[6] N. G. Leveson and C. S. Turner, "An Investigation of the Therac-25 Accidents," *Computer*, vol. 26, pp. 18–41, July 1993.

[7] N. G. Leveson, "Medical Devices: The Therac-25," in *Safeware*, pp. 515–554, Addison-Wesley, 1995.

[8] "Final Report on the August 14, 2003 Blackout in the United States and Canada: Causes and Recommendations," tech. rep., U.S.-Canada Power System Outage Task Force, April 2004.

[9] M. Dowson, "The Ariane 5 Software Failure," *SIGSOFT Softw. Eng. Notes*, vol. 22, p. 84, Mar. 1997.

[10] A. Holzer and J. Ondrus, "Mobile application market: A developer's perspective," *Telematics and Informatics*, vol. 28, no. 1, pp. 22 – 31, 2011. Mobile Service Architecture and Middleware.

[11] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A View of Cloud Computing," *Commun. ACM*, vol. 53, pp. 50–58, Apr. 2010.

[12] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787 – 2805, 2010.

[13] V. Wiels, R. Delmas, D. Doose, P.-L. Garoche, J. Cazin, and G. Durrieu, "Formal Verification of Critical Aerospace Software," *AerospaceLab*, pp. 1–8, May 2012.

[14] M. Broy, I. H. Kruger, A. Pretschner, and C. Salzmann, "Engineering Automotive Software," *Proceedings of the IEEE*, vol. 95, pp. 356–373, Feb 2007.

[15] R. N. Charette, "This car runs on code," *IEEE Spectrum*, vol. 46, no. 3, pp. 3–9, 2009.

[16] F. Cristian, "Exception Handling," in *Dependability of Resilient Computers*, pp. 68–97, Blackwell Scientific Publications, 1989.

[17] A. Romanovsky, "A Looming Fault Tolerance Software Crisis?," *SIGSOFT Softw. Eng. Notes*, vol. 32, pp. 1–4, Mar. 2007.

[18] B. Boehm and V. R. Basili, "Software Defect Reduction Top 10 List," *Computer*, vol. 34, pp. 135–137, Jan. 2001.

[19] A. Iliasov and A. Romanovsky, "Refinement Patterns for Fault Tolerant Systems," in *2008 Seventh European Dependable Computing Conference*, pp. 167–176, May 2008.

[20] P. Sacramento, B. Cabral, and P. Marques, "Unchecked exceptions: Can the programmer be trusted to document exceptions?," in *Second International Conference on Innovative Views of .NET Technologies (IVNET 2006)*, Microsoft, October 2006.

[21] J. Oliveira, D. Borges, T. Silva, N. Cacho, and F. Castor, "Do android developers neglect error handling? a maintenance-centric study on the relationship between android abstractions and uncaught exceptions," *Journal of Systems and Software*, vol. 136, pp. 1 – 18, 2018.

[22] J. Wu, S. Liu, S. Ji, M. Yang, T. Luo, Y. Wu, and Y. Wang, "Exception Beyond Exception: Crashing Android System by Trapping in "uncaughtException"," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ICSE-SEIP '17, (Piscataway, NJ, USA), pp. 283–292, IEEE Press, 2017.

[23] A. K. Maji, F. A. Arshad, S. Bagchi, and J. S. Rellermeyer, "An Empirical Study of the Robustness of Inter-component Communication in Android," in *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2012.

[24] M. Kechagia and D. Spinellis, "Undocumented and Unchecked: Exceptions That Spell Trouble," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, (New York, NY, USA), pp. 312–315, ACM, 2014.

[25] R. Coelho, L. Almeida, G. Gousios, and A. van Deursen, "Unveiling Exception Handling Bug Hazards in Android Based on GitHub and Google Code Issues," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, (Piscataway, NJ, USA), pp. 134–145, IEEE Press, 2015.

[26] M. B. Kery, C. Le Goues, and B. A. Myers, "Examining Programmer Practices for Locally Handling Exceptions," in *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, (New York, NY, USA), pp. 484–487, ACM, 2016.

[27] K. Choi and B.-M. Chang, "A Lightweight Approach to Component-level Exception Mechanism for Robust Android Apps," *Comput. Lang. Syst. Struct.*, vol. 44, pp. 283–298, Dec. 2015.

[28] R. D. Banker, S. M. Datar, and D. Zweig, "Software Complexity and Maintainability," in *Proceedings of the Tenth International Conference on Information Systems*, ICIS '89, (New York, NY, USA), pp. 247–255, ACM, 1989.

[29] M. Lippert and C. V. Lopes, "A Study on Exception Detection and Handling Using Aspect-oriented Programming," in *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, (New York, NY, USA), pp. 418–427, ACM, June 2000.

[30] M. Patterns, *Microsoft Application Architecture Guide*. Microsoft Press, 2nd ed., 2009.

[31] R. Alexandersson, P. Öhman, and M. Ivarson, "Aspect Oriented Software Implemented Node Level Fault Tolerance," in *Proceedings of the 9th IASTED International Conference on Software Engineering and Applications*, pp. 140–147, Phoenix, 2005.

[32] M. Bruntink, A. van Deursen, and T. Tourwé, "Discovering Faults in Idiom-based Exception Handling," in *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, (New York, NY, USA), pp. 242–251, ACM, 2006.

[33] A. Kolawa and D. Huizinga, *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press, 2007.

[34] R. de Lemos and A. Romanovsky, "Exception handling in a cooperative object-oriented approach," in *Object-Oriented Real-Time Distributed Computing, 1999. (ISORC '99) Proceedings. 2nd IEEE International Symposium on*, pp. 3–13, 1999.

[35] A. F. Rosene, J. E. Connolly, and K. M. Bracy, "Software Maintainability – What It Means and How to Achieve It," *IEEE Transactions on Reliability*, vol. R-30, pp. 240–245, Aug 1981.

[36] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP'97 — Object-Oriented Programming* (M. Akşit and S. Matsuoka, eds.), (Berlin, Heidelberg), pp. 220–242, Springer Berlin Heidelberg, 1997.

[37] E. L. Duke, "V&V of flight and mission-critical software," *IEEE Software*, vol. 6, pp. 39–45, May 1989.

[38] R. E. Brown, E. R. Masanet, B. Nordman, W. F. Tschudi, A. Shehabi, J. Stanley, J. G. Koomey, D. A. Sartor, and P. T. Chan, "Report to Congress on Server and Data Center Energy Efficiency: Public Law 109-431," tech. rep., Lawrence Berkeley National Laboratory, Berkeley, CA, June 2008.

[39] M. P. Mills, "The cloud begins with coal – Big data, big networks, big infrastructure, and big power," Tech. Rep. August 2013, Digital Power Group, 2013.

[40] K. Barr, *ASIC Design in the Silicon Sandbox: A Complete Guide to Building Mixed-Signal Integrated Circuits*. McGraw-Hill's AccessEngineering, McGraw-Hill Education, 2007.

[41] "The New York Times, There Is Nothing Virtual About Bitcoin's Energy Appetite." https://www.nytimes.com/2018/01/21/technology/bitcoin-mining-energy-consumption.html. Accessed: 2018-03-28.

[42] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system." https://bitcoin.org/bitcoin.pdf, March 2009. Accessed: 2018-03-28.

[43] S. Borkar, "Thousand Core Chips: A Technology Perspective," in *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, (New York, NY, USA), pp. 746–749, ACM, 2007.

[44] P. Garraghan, I. S. Moreno, P. Townend, and J. Xu, "An Analysis of Failure-Related Energy Waste in a Large-Scale Cloud Environment," *IEEE Transactions on Emerging Topics in Computing*, vol. 2, pp. 166–180, June 2014.

[45] R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming*. Greenwich, CT, USA: Manning Publications Co., 2003.

[46] W. R. Stevens, *TCP/IP Illustrated (Vol. 1): The Protocols*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1993.

[47] W. Dweik, M. Annavaram, and M. Dubois, "Reliability-aware Exceptions: Tolerating Intermittent Faults in Microprocessor Array Structures," in *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, (3001 Leuven, Belgium, Belgium), pp. 101:1–101:6, European Design and Automation Association, 2014.

[48] B. Randell, "System Structure for Software Fault Tolerance," *SIGPLAN Not.*, vol. 10, pp. 437–449, Apr. 1975.

[49] R. E. Lyons and W. Vanderkulk, "The Use of Triple-modular Redundancy to Improve Computer Reliability," *IBM J. Res. Dev.*, vol. 6, pp. 200–209, Apr. 1962.

[50] B. Randell and J. Xu, "The evolution of the recovery block concept," in *Software Fault Tolerance*, pp. 1–22, John Wiley & Sons Ltd, Sept 1994.

[51] L. Chen and A. Avižienis, "N-version programming: a fault-tolerance approach to reliability of software operation," in *Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth International Symposium on*, pp. 113–119, Jun 1995.

[52] J. Gray, "The Transaction Concept: Virtues and Limitations (Invited Paper)," in *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, pp. 144–154, VLDB Endowment, 1981.

[53] T. Haerder and A. Reuter, "Principles of Transaction-oriented Database Recovery," *ACM Comput. Surv.*, vol. 15, pp. 287–317, Dec. 1983.

[54] F. Cristian, "A Recovery Mechanism for Modular Software," in *Proceedings of the 4th International Conference on Software Engineering*, ICSE '79, (Piscataway, NJ, USA), pp. 42–50.A, IEEE Press, 1979.

[55] T. Anderson and P. Lee, *Fault tolerance, principles and practice*. Prentice/Hall International, 1981.

[56] A. DeHon, N. Carter, and H. Quinn, "Final Report for CCC Cross-Layer Reliability Visioning Study." http://relxlayer.org/, March 2011. Accessed: 2018-03-28.

[57] C. H. Ho, M. de Kruijf, K. Sankaralingam, B. Rountree, M. Schulz, and B. R. de Supinski, "Mechanisms and Evaluation of Cross-Layer Fault-Tolerance for Supercomputing," in *2012 41st International Conference on Parallel Processing*, pp. 510–519, Sept 2012.

[58] T. Arampatzis, J. Lygeros, and S. Manesis, "A Survey of Applications of Wireless Sensors and Wireless Sensor Networks," in *Proceedings of the 2005 IEEE International Symposium on, Mediterrean Conference on Control and Automation Intelligent Control, 2005.*, pp. 719–724, June 2005.

[59] L. Carnevali, L. Ridi, and E. Vicario, "Stochastic Fault Trees for cross-layer power management of WSN monitoring systems," in *2009 IEEE Conference on Emerging Technologies Factory Automation*, pp. 1–8, Sept 2009.

[60] P. Sujae and M. Vigneshpandi, "A cross layer fault tolerant communication architecture for wireless sensor networks," *Middle - East Journal of Scientific Research*, vol. 20, pp. 1292–1296, 01 2014.

[61] Y. Wang, H. Wu, and N.-F. Tzeng, "Cross-Layer Protocol Design and Optimization for Delay/Fault-Tolerant Mobile Sensor Networks (DFT-MSN's)," *IEEE Journal on Selected Areas in Communications*, vol. 26, pp. 809–819, June 2008.

[62] R. Miller and A. Tripathi, "The Guardian Model and Primitives for Exception Handling in Distributed Systems," *IEEE Trans. Softw. Eng.*, vol. 30, pp. 1008–1022, Dec. 2004.

[63] M. Blanke, R. Izadi-Zamanabadi, S. Bøgh, and C. Lunau, "Fault-tolerant control systems — A holistic view," *Control Engineering Practice*, vol. 5, no. 5, pp. 693–702, 1997.

[64] S. P. Azad, B. Niazmand, J. Raik, G. Jervan, and T. Hollstein, "Holistic Approach for Fault-Tolerant Network-on-Chip based Many-Core Systems," *CoRR*, vol. abs/1601.07089, 2016.

[65] A. Martens, C. Borchert, M. Nieke, O. Spinczyk, and R. Kapitza, "CrossCheck: A Holistic Approach for Tolerating Crash-Faults and Arbitrary Failures," in *Proceedings - 2016 12th European Dependable Computing Conference, EDCC 2016*, pp. 65–76, Sept 2016.

[66] R. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal on Robotics and Automation*, vol. 2, pp. 14–23, Mar 1986.

[67] N. J. Nilsson, "Teleo-reactive Programs for Agent Control," *Journal of Artificial Intelligence Research*, vol. 1, pp. 139–158, Jan. 1994.

[68] P. J. Prisaznuk, "Integrated modular avionics," in *Proceedings of the IEEE 1992 National Aerospace and Electronics Conference NAECON 1992*, pp. 39–45 vol.1, May 1992.

[69] Y.-H. Lee, M. Younis, and J. Zhou, "An integrated scheduling mechanism for fault-tolerant modular avionics systems," in *1998 IEEE Aerospace Conference Proceedings (Cat. No.98TH8339)*, vol. 4, pp. 21–29 vol.4, Mar 1998.

[70] C. B. Watkins and R. Walter, "Transitioning from federated avionics architectures to Integrated Modular Avionics," in *2007 IEEE/AIAA 26th Digital Avionics Systems Conference*, pp. 2.A.1–1–2.A.1–10, Oct 2007.

[71] T. A. Proebsting and S. A. Watterson, "Filter Fusion," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, (New York, NY, USA), pp. 119–130, ACM, 1996.

[72] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable, "Building Reliable, High-performance Communication Systems from Components," *SIGOPS Oper. Syst. Rev.*, vol. 33, pp. 80–92, Dec. 1999.

[73] B. Forouzan, *TCP/IP Protocol Suite*. New York, NY, USA: McGraw-Hill, Inc., 4 ed., 2010.

[74] J. Postel, "Transmission Control Protocol." RFC 793 (Standard), Sept. 1981. Updated by RFCs 1122, 3168.

[75] R. Ludwig, *Eliminating Inefficient Cross-Layer Interactions in Wireless Networking*. Phd thesis, RWTH Aahen, 2000.

[76] K. Pentikousis, "TCP in wired-cum-wireless environments," *IEEE Communications Surveys Tutorials*, vol. 3, pp. 2–14, Fourth 2000.

[77] G. Xylomenos, G. C. Polyzos, P. Mahonen, and M. Saaranen, "TCP performance issues over wireless links," *IEEE Communications Magazine*, vol. 39, pp. 52–58, Apr 2001.

[78] J. Stone and C. Partridge, "When the CRC and TCP Checksum Disagree," *SIGCOMM Comput. Commun. Rev.*, vol. 30, pp. 309–319, Aug. 2000.

[79] "Amazon S3 Availability Event: July 20, 2008." http://status.aws.amazon.com/s3-20080720.html. Accessed: 2018-03-28.

[80] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini, "Practical Hardening of Crash-Tolerant Systems," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, (Boston, MA), pp. 453–466, USENIX, 2012.

[81] "TCP Checksums Are Not Enough." http://www.evanjones.ca/tcp-checksums.html. Accessed: 2018-03-28.

[82] V. Jacobson, "Congestion Avoidance and Control," *SIGCOMM Comput. Commun. Rev.*, vol. 18, pp. 314–329, Aug. 1988.

[83] M. Adeel and A. A. Iqbal, "TCP Congestion Window Optimization for CDMA2000 Packet Data Networks," in *2007 4th International Conference on Information Technology New Generations(ITNG)*, vol. 00, pp. 31–35, April 2007.

[84] N. Möller, K. H. Johansson, and H. Hjalmarsson, "Making retransmission delays in wireless links friendlier to TCP," in *Conference on Decision and Control*, 2004.

[85] B. S. Blanchard, W. J. Fabrycky, and W. J. Fabrycky, *Systems engineering and analysis*, vol. 4. Prentice Hall Englewood Cliffs, NJ, 1990.

[86] "ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary," *ISO/IEC/IEEE 24765:2010(E)*, pp. 1–418, Dec 2010.

[87] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2nd ed., 2002.

[88] A. Avižienis, "The Dependability Problem: Introduction and Verification of Fault Tolerance for a Very Complex System," in *Proceedings of the 1987 Fall Joint Computer Conference on Exploring Technology: Today and Tomorrow*, ACM '87, (Los Alamitos, CA, USA), pp. 89–93, IEEE Computer Society Press, 1987.

[89] A. Burns and R. I. Davis, "A Survey of Research into Mixed Criticality Systems," *ACM Comput. Surv.*, vol. 50, pp. 82:1–82:37, Nov. 2017.

[90] I. Lopatkin, A. Iliasov, and A. Romanovsky, "Rigorous Development of Dependable Systems Using Fault Tolerance Views," in *2011 IEEE 22nd International Symposium on Software Reliability Engineering*, pp. 180–189, Nov 2011.

[91] F. Jahanian and A. K. Mok, "Modechart: a specification language for real-time systems," *IEEE Transactions on Software Engineering*, vol. 20, pp. 933–947, Dec 1994.

[92] F. L. Dotti, A. Iliasov, L. Ribeiro, and A. Romanovsky, "Modal Systems: Specification, Refinement and Realisation," in *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ICFEM '09, (Berlin, Heidelberg), pp. 601–619, Springer-Verlag, 2009.

[93] M. A. Iqbal, J. H. Saltz, and S. Bokhart, "Performance tradeoffs in static and dynamic load balancing strategies," March 1986.

[94] R. Hilliard, I. Malavolta, H. Muccini, and P. Pelliccione, "On the Composition and Reuse of Viewpoints across Architecture Frameworks," in *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, pp. 131–140, Aug 2012.

[95] A. Rafiev, F. Xia, A. Iliasov, R. Gensh, A. Aalsaud, A. Romanovsky, and A. Yakovlev, "Order Graphs and Cross-Layer Parametric Significance-Driven Modelling," in *Proceedings of the 2015 15th International Conference on Application of Concurrency to System Design*, ACSD '15, (Washington, DC, USA), pp. 110–119, IEEE Computer Society, 2015.

[96] A. Rafiev, F. Xia, A. Iliasov, R. Gensh, A. Aalsaud, A. Romanovsky, and A. Yakovlev, "Power-proportional modelling fidelity," in *Proceedings of the 1st Workshop on Model-Implementation Fidelity*, DATE 2015, (Grenoble, France), 2015.

[97] D. E. Perry and A. L. Wolf, "Foundations for the Study of Software Architecture," *SIGSOFT Softw. Eng. Notes*, vol. 17, pp. 40–52, Oct. 1992.

[98] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. SEI series in software engineering, Addison-Wesley, 2003.

[99] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, "Abstractions for Software Architecture and Tools to Support Them," *IEEE Trans. Softw. Eng.*, vol. 21, pp. 314–335, Apr. 1995.

[100] D. Garlan and M. Shaw, "An Introduction to Software Architecture," tech. rep., Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[101] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[102] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[103] Clements, P. and Bachmann, F. and Bass, L. and Garlan, D. and Ivers, J. and Little, R. and Merson, P. and Nord, R. and Stafford, J., *Documenting Software Architectures: Views and Beyond*. SEI Series in Software Engineering, Pearson Education, 2010.

[104] N. Medvidovic and R. N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Trans. Softw. Eng.*, vol. 26, pp. 70–93, Jan. 2000.

[105] R. Allen and D. Garlan, "A Formal Basis for Architectural Connection," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, pp. 213–249, July 1997.

[106] G. Booch, J. Rumbaugh, and I. Jacobson, *Unified Modeling Language User Guide, The (2Nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[107] A. Rafiev, F. Xia, A. Iliasov, R. Gensh, A. Aalsaud, A. Romanovsky, and A. Yakovlev, "Selective abstraction and stochastic methods for scalable power modelling of heterogeneous systems," in *2016 Forum on Specification and Design Languages (FDL)*, pp. 1–7, Sept 2016.

[108] W. H. Sanders and J. F. Meyer, "Stochastic Activity Networks: Formal Definitions and Concepts," in *Lectures on Formal Methods and Performance Analysis* (E. Brinksma, H. Hermanns, and J.-P. Katoen, eds.), pp. 315–343, New York, NY, USA: Springer-Verlag New York, Inc., 2002.

[109] Z. Bubnicki, *Modern Control Theory*. Springer-Verlag Berlin Heidelberg, 2005.

[110] K. Ogata, *Modern Control Engineering*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 5th ed., 2009.

[111] A. Movaghar, *Performability Modeling with Stochastic Activity Networks*. PhD thesis, University of Michigan, Ann Arbor, MI, USA, 1985. AAI8520952.

[112] J. F. Meyer, A. Movaghar, and W. H. Sanders, "Stochastic Activity Networks: Structure, Behavior, and Application," in *International Workshop on Timed Petri Nets*, (Washington, DC, USA), pp. 106–115, IEEE Computer Society, 1985.

[113] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981.

[114] W. H. Sanders and J. F. Meyer, "METASAN: A Performability Evaluation Tool Based on Stochastic Activity Networks," in *Proceedings of 1986 ACM Fall Joint Computer Conference*, ACM '86, (Los Alamitos, CA, USA), pp. 807–816, IEEE Computer Society Press, 1986.

[115] W. Sanders, W. Obal, M. Qureshi, and F. Widjanarko, "The ultrasan modeling environment," *Performance Evaluation*, vol. 24, no. 1, pp. 89 – 115, 1995. Performance Modeling Tools.

[116] W. H. Sanders, "Integrated frameworks for multi-level and multi-formalism modeling," in *Proceedings 8th International Workshop on Petri Nets and Performance Models (Cat. No.PR00331)*, pp. 2–9, 1999.

[117] "The Möbius modelling tool." https://www.mobius.illinois.edu. Accessed: 2018-03-28.

[118] R. Mansharamani, "An overview of discrete event simulation methodologies and implementation," *Sadhana*, vol. 22, pp. 611–627, Oct 1997.

[119] F. W. Zurcher and B. Randell, "Iterative Multi-Level Modeling - A Methodology for Computer System Design," in *in Proceedings IFIP Congress 1968*, pp. 138–142, Press, 1968.

[120] A. Ehrenfeucht and G. Rozenberg, "Zoom structures and reaction systems yield exploration systems," *International Journal of Foundations of Computer Science*, vol. 25, no. 03, pp. 275–305, 2014.

[121] A. Rafiev, A. Iliasov, A. Romanovsky, A. Mokhov, F. Xia, and A. Yakovlev, "ArchOn: Architecture-open Resource-driven Cross-layer Modelling Framework," in *Proceedings of International Workshop on Engineering Simulations for Cyber-Physical Systems*, ES4CPS '14, (New York, NY, USA), pp. 21:21–21:24, ACM, 03 2014.

[122] M. Tamiz and D. F. Jones, "Goal programming and Pareto efficiency," *Journal of Information and Optimization Sciences*, vol. 17, no. 2, pp. 291–307, 1996.

[123] N. Metropolis and S. Ulam, "The Monte Carlo Method," *Journal of the American Statistical Association*, vol. 44, no. 247, pp. 335–341, 1949. PMID: 18139350.

[124] Itseez, "Open Source Computer Vision Library." https://github.com/itseez/opencv, 2017.

[125] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal: Software Tools for the Professional Programmer*, vol. 25, no. 11, pp. 120–123, 2000.

[126] R. Smith, "An Overview of the Tesseract OCR Engine," in *Proceedings of the Ninth International Conference on Document Analysis and Recognition - Volume 02*, ICDAR '07, (Washington, DC, USA), pp. 629–633, IEEE Computer Society, Sept 2007.

[127] D. L. Baggio, S. Emami, D. M. Escrivá, K. Ievgen, N. Mahmood, J. Saragih, and R. Shilkrot, *Mastering OpenCV with Practical Computer Vision Projects*. Birmingham: Packt Publishing, Limited, 2012.

[128] J. V. Guttag, *Introduction to Computation and Programming Using Python*. The MIT Press, 2013.

[129] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003.

[130] B. Meyer, *Object-Oriented Software Construction*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1st ed., 1988.

[131] R. C. Martin, "The Open-closed Principle," in *More C++ Gems* (R. C. Martin, ed.), pp. 97–112, New York, NY, USA: Cambridge University Press, 2000.

[132] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured Design," *IBM Systems Journal*, vol. 13, pp. 115–139, June 1974.

[133] E. W. Dijkstra, *On the Role of Scientific Thought*, pp. 60–66. New York, NY: Springer New York, 1982.

[134] W. B. Frakes and K. Kang, "Software Reuse Research: Status and Future," *IEEE Trans. Softw. Eng.*, vol. 31, pp. 529–536, July 2005.

[135] N. Cacho, *Supporting Maintainable Exception Handling with Explicit Exception Channels*. Phd thesis, Computing Department, Lancaster University, 2009.

[136] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," in *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, (London, UK, UK), pp. 327–353, Springer-Verlag, 2001.

[137] F. C. Filho, A. Garcia, and C. M. F. Rubira, "Extracting Error Handling to Aspects: A Cookbook," in *2007 IEEE International Conference on Software Maintenance*, pp. 134–143, Oct 2007.

[138] S. Karol, N. A. Rink, B. Gyapjas, and J. Castrillon, "Fault Tolerance with Aspects: A Feasibility Study," in *Proceedings of the 15th International Conference on Modularity*, MODULARITY 2016, (New York, NY, USA), pp. 66–69, ACM, 2016.

[139] R. Alexandersson, P. Öhman, and J. Karlsson, *Aspect-Oriented Implementation of Fault Tolerance: An Assessment of Overhead*, pp. 466–479. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[140] N. Cacho, F. Dantas, A. Garcia, and F. Castor, "Exception Flows Made Explicit: An Exploratory Study," in *Proceedings of the 2009 XXIII Brazilian Symposium on Software Engineering*, SBES '09, (Washington, DC, USA), pp. 43–53, IEEE Computer Society, 2009.

[141] P. Maes, "Concepts and Experiments in Computational Reflection," *SIGPLAN Not.*, vol. 22, pp. 147–155, Dec. 1987.

[142] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1 ed., 2008.

[143] A. Vajda, *Programming Many-Core Chips*. Springer Publishing Company, Incorporated, 1st ed., 2011.

[144] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), (New York, NY, USA), pp. 483–485, ACM, 1967.

[145] "big.LITTLE technology ARM.com." https://developer.arm.com/technologies/big-little. Accessed: 2018-03-28.

[146] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu, "Energy types," *SIGPLAN Not.*, vol. 47, pp. 831–850, Oct. 2012.

[147] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, pp. 476–493, June 1994.

[148] M. Ceccato and P. Tonella, "Measuring the Effects of Software Aspectization," in *Proceedings of the 1st Workshop on Aspect Reverse Engineering*, (Delft University of Technology, the Netherlands), pp. 1–5, November 2004.

[149] D. Coleman, D. Ash, B. Lowther, and P. Oman, "Using Metrics to Evaluate Software System Maintainability," *Computer*, vol. 27, pp. 44–49, Aug. 1994.

[150] I. Heitlager, T. Kuipers, and J. Visser, "A Practical Model for Measuring Maintainability," in *Proceedings of the 6th International Conference on Quality of Information and Communications Technology*, QUATIC '07, (Washington, DC, USA), pp. 30–39, IEEE Computer Society, Sept 2007.