COMPUTING
SCIENCE

Title: Experience Report: Evaluation of Holistic Fault Tolerance

Names: Rem Gensh, Alexander Romanovsky, Alessandro Garcia

TECHNICAL REPORT SERIES

No. CS-TR- 1507 – May 2017

TECHNICAL REPORT SERIES

No. CS-TR- 1507 Date 15th May 2017

Title: Experience Report: Evaluation of Holistic Fault Tolerance

Authors: Rem Gensh, Alexander Romanovsky, Alessandro Garcia

Abstract: 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 made on the functional features of the application, whereas fault tolerance mechanisms are neglected and as a result do not provide sufficient maintainability and reusability. In our previous work we introduced the concept of Holistic Fault Tolerance as 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. This paper evaluates the maintainability of the Holistic Fault Tolerance architecture using experimental analysis of the developer's effort required to implement various modifications of the fault tolerance functionality. The paper starts by justifying the choice of modifications and evaluation techniques. Then the aspect-oriented implementation we proposed for Holistic Fault Tolerance is evaluated by conducting its experimental comparison with a standard object-oriented fault tolerance implementation. The evaluation shows that the implementation with Holistic Fault Tolerance makes fault tolerance mechanisms easier to maintain and ensures higher modularity of the source code.

Printed and published by Newcastle University,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical Details

Title and Authors

Experience Report: Evaluation of Holistic Fault Tolerance
Rem Gensh, Alexander Romanovsky, Alessandro Garcia

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR 1507

Abstract

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 made on the functional features of the application, whereas fault tolerance mechanisms are neglected and as a result do not provide sufficient maintainability and reusability. In our previous work we introduced the concept of Holistic Fault Tolerance as 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. This paper evaluates the maintainability of the Holistic Fault Tolerance architecture using experimental analysis of the developer's effort required to implement various modifications of the fault tolerance functionality. The paper starts by justifying the choice of modifications and evaluation techniques. Then the aspect-oriented implementation we proposed for Holistic Fault Tolerance is evaluated by conducting its experimental comparison with a standard object-oriented fault tolerance implementation. The evaluation shows that the implementation with Holistic Fault Tolerance makes fault tolerance mechanisms easier to maintain and ensures higher modularity of the source code.

About the authors

Rem Gensh is currently a Research Technician in the Secure & Resilient Systems group, School of Computing Science at Newcastle University. He graduated at Kyrgyz-Russian Slavic University, Kyrgyzstan, with honors in 2008. After a 6-years extensive industrial experience as a software developer and a team leader, he reallocated to research area and started his PhD in 2014. He is involved in EPSRC/UK PRiME project. His research interests include fault tolerance, energy-efficient software design and many-core architectures.

Alexander (Sascha) Romanovsky is a Professor in the Centre for Software and Reliability, Newcastle University. He is the leader of the Dependability Group

at the School of Computing Science. His main research interests are system dependability, fault tolerance, software architectures, exception handling, error recovery, system structuring and verification of fault tolerance.He received a M.Sc. degree in Applied Mathematics from Moscow State University and a PhD degree in Computer Science from St. Petersburg State Technical University. He was with this University from 1984 until 1996, doing research and teaching. In 1991 he worked as a visiting researcher at ABB Ltd Computer Architecture Lab Research Center, Switzerland. In 1993 he was a visiting fellow at Istituto di Elaborazione della Informazione, CNR, Pisa, Italy. In 1993-94 he was a post-doctoral fellow with the Department of Computing Science, University of Newcastle upon Tyne. Alexander is now the Principle Investigator of the TrAmS-2 EPSRC/UK platform grant on Trustworthy Ambient Systems (2012-16) and of the EPSRC/RSSB research project SafeCap on Overcoming the Railway Capacity Challenges without Undermining Rail Network Safety (2011-14), and the Co-investigator of the EPSRC PRiME program grant (2013-18) and of the FP7 COMPASS Integrated Project (2011-14).

Alessandro Garcia is Associate Professor at the Informatics Department, PUC-Rio, Brazil. He is the Head of the Opus Research Group. Alessandro's research interests include Exception Handling, Software Fault Tolerance, Multi-Agent Systems, Software Architecture, Aspect-Oriented Software Development, Computational Reflection, Design Patterns and Software Metrics.

Suggested keywords

fault tolerance; maintainability; maintenance; software evaluation; aspect oriented programming;

Experience Report: Evaluation of Holistic Fault Tolerance

Rem Gensh, Alexander Romanovsky
Centre for Software Reliability
Newcastle University
Newcastle upon Tyne, UK
{r.gensh, alexander.romanovsky}@newcastle.ac.uk

Alessandro Garcia
Informatics Department
PUC-Rio
Rio de Janeiro, Brazil
afgarcia@inf.puc-rio.br

***Abstract*—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 made on the functional features of the application, whereas fault tolerance mechanisms are neglected and as a result do not provide sufficient maintainability and reusability. In our previous work [1] we introduced the concept of Holistic Fault Tolerance as 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. This paper evaluates the maintainability of the Holistic Fault Tolerance architecture using experimental analysis of the developer's effort required to implement various modifications of the fault tolerance functionality. The paper starts by justifying the choice of modifications and evaluation techniques. Then the aspect-oriented implementation we proposed for Holistic Fault Tolerance is evaluated by conducting its experimental comparison with a standard object-oriented fault tolerance implementation. The evaluation shows that the implementation with Holistic Fault Tolerance makes fault tolerance mechanisms easier to maintain and ensures higher modularity of the source code.**

Keywords—fault tolerance; maintainability; maintenance; software evaluation; aspect oriented programming;

I. INTRODUCTION

Software maintainability is central in reducing maintenance costs and decreasing the downtime in case of system modification or, at worst, in case of system failure. However, it often happens that maintenance actions unintentionally introduce new bugs and faults due to system complexity. To avoid or at least minimise such occurrences, modules of the system should not be significantly 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 practice is usually followed when implementing the functional system features, such as business logic or data access. The situation with non-functional features is, however, different. The source code responsible for diagnostics, security or fault tolerance (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 are related to 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.

In our previous studies, we proposed a vision of Holistic Fault Tolerance (HFT) [1] and a detailed description of the HFT architecture [2]. The former pursued two main goals. First, it would allow developers to design and maintain complex reliable applications in a more efficient fashion than the conventional structuring techniques by supporting a disciplined and systematic way of capturing and modularising the cross-cutting functionalities related to error detection and error recovery. 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.). In this paper, we address the first goal and, more specifically, report on the evaluation of HFT maintainability.

In this study, we provide an experimental evaluation of FT code maintainability in a software application with the HFT architecture. This architecture was implemented using Aspect-Oriented Programming (AOP), more specifically an AspectJ AOP extension of Java language [3]. To evaluate the HFT approach, we implemented two versions of the same application. The first one is based on the HFT architecture with AOP, whereas the second was implemented using standard object-oriented programming (OOP) techniques. After that we carried out a set of experiments on both versions of the application. During the 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.

In this work, we mainly focus on small and medium scale software applications; this is assumed when we reason about the HFT architecture. Our ongoing work is to extend the HFT scalability to ensure that HFT can be applied to a large-scale

This work is supported by the EPSRC PRiME EP/K034448/1 and MRC Newton001 MR/M026388/1 projects.

system, encompassing different layers of the system stack, from hardware to software. However, in this case the HFT architecture would require more complex design and implementation, which are out of scope of this paper.

The HFT approach does not involve the introduction of new or the change of the existing well established techniques [4]. The idea is to support reasoning about the system FT at the system level rather than at the level of individual units and apply the error detection and recovery techniques at system level to make FT-related design more maintainable, modular and reusable. We should note here that FT and maintainability concepts are closely interconnected. FT is a means of dependability, whereas maintainability as an attribute of dependability [5]. The HFT approach focuses on both concepts to ensure dependable operation of the system.

The main contributions of this paper are:

- An experimental evaluation of the HFT architecture.
- The AOP implementation of the HFT architecture.

The rest of the paper is organised as follows. Section 2 provides the background of the relevant areas. The HFT architecture and its AOP implementation is described in Section 3. The experiment setting is explained in Section 4. Evaluation and discussion of the obtained results is provided in Section 5. Concluding remarks and future plans are given in Section 6.

II. BACKGROUND

In this section we discuss the basic concepts of dependability and the main principles of high-quality software engineering. We analyse the existing approaches to evaluating the reusability, modularity and changeability of program code. The state-of-the-art studies of the centralised FT management and approaches to using AOP for the implementation of the system FT are examined and compared with the HFT approach.

A. Taxonomy of dependability

The main concepts and the taxonomy of dependability are introduced in [4]. *Dependability* is defined as "the ability of a system to avoid service failures that are more frequent or more severe that is acceptable". *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 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.

B. Principles of software structural quality

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

Abstraction is applied to deal with complexity of a computer system. It allows the developer to work with data objects without going to their implementation details. Each object provides a simple interface, while intricate details of the implementation are encapsulated.

According to *Single Responsibility Principle* [6] 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. If the module has more than one reason of change then it should be split in two or more modules.

Open/closed principle [7, 8] 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 the changes in dependent modules. Moreover, OCP guarantees that the 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 an existing code.

Coupling and *cohesion* usually are considered together [9]**.** The former describes interdependencies 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 FT functionality of the system should be placed in a separate module rather than partially implemented by each individual module.

Separation of concerns (SoC) [10] is a design principle assuming that 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, API for external clients. However, some concerns are dispersed across different part of the program. These *crosscutting concerns* affect the entire system and cannot be distinguished straightforwardly. Various information loggers are a typical example of crosscutting concerns. In object-oriented design these concerns can create high degree of tangling and affect modularity of the program. AOP is applied to assist in the separation of crosscutting concerns by encapsulating them into aspects. There are academic and industrial studies [11] 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. Throughout this paper we consider system-wide FT as a crosscutting concern.

Code reuse principle implies that it is good practice to use an existing functions, patterns and modules in order to reduce redundancy, decrease development time and improve maintainability.

C. Existing holistic approaches to fault tolerance

A three-layer architecture for the FT control is introduced in [12]. 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 autonomous supervisor from the third layer. To achieve high availability and avoid the system failure, authors prefer to apply reconfiguration of the system after fault detection rather than increased robustness with performance overheads. In this work authors do not consider separate modules that are responsible for performance monitoring and error handling.

System Health Monitoring Unit is used by network-on-chip many-core system [13]. This unit has a holistic view of the health status of the system components. Mapper/Scheduler Unit generates mapping and scheduling solutions for each fault configuration. This approach is bound to the specific network-on-chip architecture and may be not suitable for other architectures such as software applications

Study [14] aims at providing high availability for the request-oriented distributed system using 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 the difference is detected by voting, the faulty replica is recovered by the special recovery message. AOP is applied for the protection of critical state-objects to deal with arbitrary state corruption. Experiments proved low performance overhead of this solution. The CrossCheck intended to optimise performance, but in comparison with the HFT approach it does not consider the tradeoff between reliability and performance.

D. Metrics of for the source code evaluation

A number of metrics for the object-oriented design are proposed in [15]. 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 class should be divided into subclasses to reduce complexity of the original class).

Metrics of AOP code are considered in [16]. These metrics comprise OOP metrics and AOP-specific metrics, such as number of modules affected by the given aspect (Crosscutting Degree of an Aspect), number of aspects whose advices could be triggered by operations in a given module (Coupling on Advice Execution).

E. AOP for the implementation of fault tolerance

Regarding the separation of crosscutting concerns AOP is applied to improve modularity because the same entities will be placed in one module (or aspect in AspectJ). There has been research that showed feasibility and benefits of the centralisation of 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. In [17] 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 from the normal application code making that the changes in the AOP code will be less intrusive and much simpler. However, the limitation of AOP is that it is not possible to represent global properties of exception control flows. In addition, there are not usable abstractions for composition and reusing of pluggable exception handlers.

Paper [18] provides an analysis of the claim that AOP facilitates the modularisation of exception handling mechanisms. Authors state that 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 the system with bad architecture. However, it is able to facilitate the structure of well designed systems by separating normal and exceptional activities of the system. Two main contributions of the paper are based on an interplay between AOP and error handling. The first is classification of exception handling code in terms of factors that make influence on aspectisation. The second is analysis of interactions amongst these factors.

Feasibility and evaluation of using AOP for software implemented hardware FT (SIHFT) is presented in [19]. Authors offer 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 the programs with SIHFT. The authors focus mainly on hardware FT that is implemented in software, however they do not consider FT of the entire system. In addition, this approach does not assume centralised coordination.

Paper [20] estimates the impacts 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 was 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. Authors analysed the pros and cons of the 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 significant performance overheads for systematic implementations. 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 experiments evaluating the advantages and disadvantages of explicit exception flows and implicit exception flows using three different exception handling mechanisms based on Java, AspectJ and EJFlow are presented in [21]. AspectJ provides a way to distinguish normal and error handling code but only syntactically (not semantically). In turn, the EJFlow exception handling mechanism introduces two notions: explicit exception channels and pluggable handlers. An explicit exception channel abstracts the flow of exception from the rising site to the handling site, whereas a pluggable handler is a special exception handler that could be bound to methods, classes and packages. The experiments showed that exception channels and pluggable handlers provide more robust and flexible exception handling. Therefore, the EJFlow abstractions facilitate software maintainability and make exception control flow more understandable.

Analysis of these studies showed that it is feasible to use AOP for the implementation of FT. However, the developer should carefully pick up the advices, which will be executed when certain join point is reached.

III. HFT ARCHITECTURE AND IMPLEMENTATION

In our previous studies [1], [2] we introduced the concept of the HFT and provided the architectural pattern for the design of the HFT. In this section, we focus on the implementation details of each element of the HFT architecture with assistance of AOP.

The typical software application based on the HFT architecture consists of a number of functional components satisfying functional requirements and the HFT part, which is responsible for the dependable and efficient operation of the application. The HFT part includes the HFT controller and several HFT agents.

A. HFT controller

The HFT controller is the central element of the HFT architecture. It coordinates system-wide FT strategies and distributes available computer resources among the application components. In addition, it performs reconfiguration of the application components if it detects that application can operate faster or more reliably. The HFT controller consist of three parts: static data storage, dynamic data storage and decision maker. Static data represents predefined HFT policies. That includes expected application performance and reliability, fault assumptions of the application (expected errors), available system resources, general structure of the application components and conditions for application reconfiguration. Dynamic data is the information about current system state, which includes performance characteristics of the application components, error rates in critical functions and diagnostic information. This data is supplied by the HFT agents. Decision maker is responsible for reconfiguration and fault handling in the application. Moreover, it chooses the most suitable error recovery action for the error in the application component that should be handled holistically. These decisions are made based on static and dynamic data.

B. HFT agent

The HFT agent in 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 application components. The HFT agent monitors, and if needed, intervenes in the control flow of critical functions in application components. The HFT agent consists of monitoring logic, intervention logic and local decision maker. Monitoring logic defines which members of the application components will be monitored by the agent. Monitoring does not involve any changes of the state inside the application components. Intervention logic determines how the control flow inside monitored functions will be affected by the HFT. Local decision maker of the HFT agent communicates with the HFT controller. Local decision maker distinguishes the data that should be transmitted to the HFT controller and the data that can be processed locally. Some HFT agents do not implement intervention logic if they perform only monitoring actions, for example performance monitoring or function calls counter.

C. The HFT controller and the HFT agetns

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. Data mapping to the HFT controller format is necessary to avoid the tangling of the HFT controller with encapsulated details of monitored components. Otherwise, the scalability of HFT controller will be deteriorated. To improve performance and avoid bottlenecks in the HFT controller, the HFT agents should filter the information and send only an important data.

D. The HFT agents and application components

The HFT agents are aware about the inner structure and encapsulated implementation details of the monitored application components, however the application components are implemented without knowledge about the HFT agents. On the one hand, this approach violates abstraction and encapsulation principles because the HFT agent is significantly dependent on the structure of monitored component. But from other hand, it assists in the separation of crosscutting concerns. We do not offer to use the HFT agents to amend functional behaviour of the application component. Instead, we propose to use the HFT agents to simplify the management of crosscutting concerns. Thus, the problem of implicit coupling between the HFT agent and monitored component is overlapped by better modularity allowing the developer to avoid code tangling and improve the understanding of the application FT techniques.

E. The HFT controller and application components

In the application with the HFT architecture, some application components should provide interfaces for 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 application components, since it uses only a predefined interface and it is aware only of general structure of the application. This link between the HFT controller and application components is supposed to be used only asynchronously. Decision maker of the HFT controller is operating in a separate thread. Based on the information from the HFT agents, it can detect that operation of the application could be more efficient or more reliable. In this case, the HFT controller sets the most suitable configuration for the application component. If the application 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 only demonstrate maintainability benefits. Reconfiguration is only available for those components of the application that provide some redundancy in their implementation. The HFT approach assumes that acceptable frequency and severity of the service failures could vary depending on user requirements or current system settings.

F. Usage of the HFT architecture

Fig. 1 illustrates an abstract application based on the HFT architecture. This application has seven functional components (C1 – C7) that implement functional requirements of the application 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 application and it is useful to monitor their inner operation to reason about the state of the entire application. 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. Dependency relation between the HFT agents and application component is implicit for the component. Thus, the application components do not know about the HFT agent. Components from the third group (C5) are not monitored by the HFT agents, but provide the 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 the HFT agents. The fourth group (C6 and C7) of components is not directly affected by the HFT architecture. This is impractical to connect all components of the application to the HFT part. It is necessary to choose only those components, which affect the operation of the entire application and are able to provide important information about the application state.

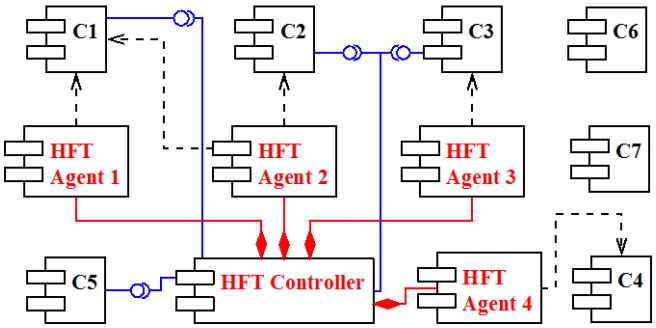

Fig. 1. The system based on the HFT architecture.

Various HFT agents can be applied in the application architecture depending on the type, size and requirements of the application. Typical examples of the HFT agents are: Error Handling Agent, Performance Agent, Diagnostics Agent.

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 [5]. The complexity of this implementation depends on application criticality. Apart from this option, we insist on the implementation of the "default" application behaviour for the case when the HFT controller or one of the agent fails. In this case the application would operate not optimally and possibly less reliable, but the failure will be avoided. The main thing is detection of the problems with the HFT elements and well-timed correction to return the application to the efficient operation.

G. AOP for the implementation of the HFT

The developer can choose various options for the implementation of the HFT, such as static classes, computational reflection and AOP. All them have advantages and disadvantages. Static classes could be more understandable for the developers that do not have an experience of working with reflection and AOP. However, the implementation of the HFT with static classes do not solve the problem with code tangling. Computational reflection can facilitate modularisation of the program and provide the same benefits for software maintainability as AOP. However, reflection works in runtime and could significantly affect performance of the application.

AOP is intended to increase modularity of the applications by the separation of crosscutting concerns. This is done by the extensions of program code behaviour in certain points of the application. At the same time the code itself is not modified since new behaviour is coded at separate modules – aspects. We propose to apply AOP for the implementation of the HFT architecture because AOP facilitates modularisation of the HFT agents, which is a basic requirement for the HFT architecture. Moreover, AOP has the benefit of composing the HFT agents and components of the application at compile time, but not in runtime. In the majority of cases "around" advice [3] would be the most suitable option for the implementation of monitoring and intervention logic of the HFT agents. For example, the

performance monitoring of the crucial function could be implemented as shown in Fig. 2. Holistic error handling is shown in Fig. 3. It should be noted that functional behaviour of the critical function is not tangled with performance monitoring and error handling. The developer should choose the functions that will be monitored with around advice. It is practical to choose those functions, which significantly affect performance and reliability of the application. This approach makes the function more readable and easily understandable.

```
CriticalFunctionResult around(CriticalClass mainLogic,
     CriticalFunctionArguments arguments) :
     Pointcuts.criticalPointcut(mainLogic, arguments){

  long startTime = System.currentTimeMillis();
  CriticalFunctionResult result =
       proceed(mainLogic, arguments);
  long execTime = System.currentTimeMillis() - startTime;
  hftController.updatePerformanceInfo(execTime);
  return result;
}
```

Fig. 2. Performance monitoring advice.

```
CriticalFunctionResult around(CriticalClass mainLogic,
     CriticalFunctionArguments arguments)
     throws Exception :
     Pointcuts.criticalPointcut(mainLogic, arguments){

  int attemptNumber = 0;
  while(true) {
    try {
     attemptNumber++;
     CriticalFunctionResult result =
          proceed(mainLogic, arguments);
     return result;
    }
    catch (Exception ex) {
      if (attemptNumber >= NumberOfAttempts)
        return CriticalFunctionResult.GetEmptyResult();
      RecoveryAction ra =
      hftController.getRecoveryAction(ex, attemptNumber);
      if (ra == RecoveryAction.Retry)
        continue;
      else if (ra == RecoveryAction.Skip)
        throw exception;
      else if (ra == RecoveryAction.TryNextAlgorithm)
        return mainLogic.alternateFumction(arguments);
    }
  }
}
```

Fig. 3. Error hanling advice.

The main benefit of the HFT architecture is centralised access to crosscutting functionality as performance adjusting and error handling. Majority of maintenance changes relating to these features will be made in corresponding aspect that is applied for the implementation of the HFT agent. HFT architecture assist 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.

IV. EXPERIMENTS

In section 2 we considered the studies describing the usage of centralised FT mechanisms and applying the AOP for the implementation of the FT. In many cases, FT is addressed as a crosscutting concern of the application that is why it should be separated from functional modules to improve the modularity. We claim that the 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 runtime. However, the counterargument is that the 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, 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 work is to gain empirical knowledge of the positive and negative effects of the HFT architecture on the software maintainability. To show the feasibility of applying the HFT architecture we carried out the experiments evaluating the HFT architecture. During these experiments, we performed a longitudinal study and analysed the effects of the HFT on software maintainability.

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 with the AOP approach, whereas the second version uses only the standard object-oriented approach. Although the implementations are different, these applications are functionally identical. For simplicity, we call the AOP-based application as the HFT-version and the OO-based application as the non-HFT version.

We have chosen this type of evaluation since the OOP is very wide-spread in modern software development. Thus, we decided to compare the maintainability of the "standard" solution implemented in OOP-style and proposed solution that is still OOP-based, but with the HFT functionality implemented with AOP. We designed the experiments to understand the challenges that could be faced by the 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 popular OOP approach. The modifications chosen for the experiments represent typical changes and bug fixes in the FT mechanisms of the medium scale software application during maintenance works. The evaluation should show, how significantly the source code of both applications is affected by each modification. In addition, it should show how easy is it to find the place (class and function) 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 of 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.

A. *Experimental setup*

We have chosen the application for the recognition of the UK number plates [2]. The functional part of both versions is

the same. The UML diagrams of both applications are shown in Fig. 4 and Fig. 5. The application receives the set of images as an input for recognition. After that the images are sent to the Initial Image Processing (IIP) component where these images are processed concurrently. This component makes initial processing of each image and tries to find the number plate area on the image. There are two algorithms for this task: rectangle detection based on OpenCV and HAAR cascade [22]. If the number plate is found it is cut from the image and sent to Number Plates Queue (NPQ). The Optical Character Recognition (OCR) component checks the NPQ and if it is not empty the OCR component takes the number plate cutout and performs the recognition of the number plate. The OCR component has two algorithms for this: Tesseract [23] and number plate recognition algorithm described in [24]. The former algorithm recognizes the entire string, while the latter algorithm requires to separate the symbols of the number plate string before the recognition.

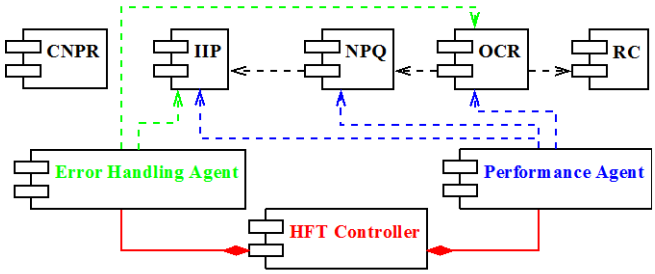

Fig. 4. Case study (the HFT version) application.

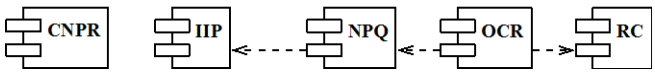

Fig. 5. Case study (the non-HFT version) application.

TABLE I. FAULT ASSUMPTIONS OF THE APPLICATION

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

Since we use third party algorithms, we cannot be sure that the functions from these libraries will not fail. One problem is impossibility to detect the required data on the image, which is not unusual situation for the image recognition operations. Another problem is exception in the third-party library. Redundant algorithms for IIP and OCR stages were introduced to deal with these problems. These fault assumptions (Table 1) are addressed differently in two applications. In the HFT version, there are the HFT controller, the Performance agent and the Error Handling agent. In the non-HFT version the FT mechanisms are distributed across the application components and do not have a centralised controller. Redundancy of the algorithms is applied for fault handling and error handling.

B. Metrics of maintainability evaluation

The following metrics of the maintainability evaluation were chosen: lines of code affected, functions affected, classes affected. Each metric consists of three parts. The first is *Quantity of Added* (lines of code, functions, classes). This metric is less critical and more preferable since we only added new functionality and we did not change 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 modifications in the places where deleted functions were called.

C. Modifications of the applications

When both versions were ready we defined and implemented a number of modifications related to the FT functionality. During the experiments, we tried to avoid the changes in the functional part of the application since the main goal of these experiments is to evaluate the maintainability of the FT-related mechanisms in the HFT architecture. Modifications that were made in both version of the application are described below.

1) Changes in the settings that are used in FT mechanisms.

Sometimes, the setting that define behaviour of FT mechanisms should be updated. We decided to check whether there is a difference between the HFT and the non-HFT versions.

2) Centralisation of thread management for IIP and OCR components.

In some software applications, there is no a special module for thread management since a chosen framework is sufficient for thread management without the effort from the developer. However, if the application is multithreaded and performs a lot of concurrent operations then it is inevitable that it will require 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 among other application components. In the non-HFT version, a new class was introduced. Main motivation for this modification is the separation of concerns, since it is not the task of the functional component to deal with thread allocation and distribution.

3) Handling of injected CPU error.

In the case study presented in [2] we introduced the CPU error to analyse the feasibility of handling hardware errors and

exceptions at the software layer with the HFT architecture. This error is not real CPU error. It is injected with specified rate inside critical functions of IIP and OCR components. In the given modification, we decided to consider two options regarding the CPU error. The first option is to introduce the CPU error to both applications and to apply system-wide action for the recovery of the CPU error. Before this modification either version did not have the any traces of the CPU error. In the HFT version we added handling of the CPU error to 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 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 became hidden for the external modules and will be recovered locally.

4) Logging diagnostics information.

For this modification, we made the changes 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 we added Diagnostics Agent, which is implemented as an aspect. For all functions in the application, which should be logged, we added corresponding *before* or *after* advice. All the modifications are concentrated in the single aspect. For the non-HFT version we added static class *Logger*. The class itself is much shorter than diagnostics aspect. However, we had to modify all the functions, whose calls should be logged. Switching off the diagnostics information is implemented approximately equally. However, if we need to remove the diagnostics, for the HFT version, we will need to delete or modify only one aspect, whereas for the non-HFT version we will need to modify a set of functions by deleting the lines of code with Logger class calls.

5) Reconfiguration logic based on operation mode.

Operation mode is a flexible option intended to provide different quality of the 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 as applicable for the 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 are 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.

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 the error. This modification involves the detection of the error by checking the result of two functions. Low quality of the result in IIP component and consequent low probability of successful recognition in the OCR component are considered as an error. It should be noted that these two conditions separately are not supposed as errors. Error recovery involves re-processing of the image with better settings at the IIP component. In the HFT version Error Handling Agent has the trace of the processing for each image. Thus, the detection for the given situation can be added at OCR stage. Error recovery action is requested from the HFT controller. For the non-HFT version we added the information about image processing steps to the object that stores the image.

V. EVALUATION AND DISCUSSION

In this section, we discuss the results of the conducted experiments that measured a set of realistic modifications in the FT-related code. These experiments show advantages and limitations of the HFT architecture with respect to the modifications of crosscutting concerns related to fault tolerance.

According to the open/closed principle [7, 8] 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 we evaluate the modification in our experiments. Experimental data is presented in table 2 (the HFT version) and table 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 with very expected result. 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.

Thread management modification metric did not show significant differences between 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 not difference between two versions is that the code managing the threads was already well structured. After modification, this code was placed to the 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 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, we do not have 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 do 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. When the developer has 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 allowance for the rate of failed operations.

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. We added only 7 LoCs in the HFT version, whereas the non-HFT version required 32 LoCs. Moreover, two functions were added to the non-HFT version. The reason of the success of the HFT version is that it already had a centralised mechanism for handling various errors. Error handling is performed by Error Handling Agent, which requests suitable recovery action at the HFT controller. So, we just added information about the CPU error to this centralised handler. In the non-HFT version we implemented CPU error handler 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 we deleted 10 LoCs. In addition, we deleted 2 functions in the non-HFT version. All other metrics are 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 the HFT is opposite. The problem for the HFT architecture here is that CPU exception was handled by Error Handling Agent and by the HFT controller. The change required to delete all this code and handle the exception inside IIP and OCR classes. We will get the benefits if we allow the class 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 of the diagnostics information is an important part the computer 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 the 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 in the system. Thus, there is a need for textual separation of the functional and diagnostics code. In addition, there should be the 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 modifications of the functional code. In the OOP approach, it is necessary to add calls to special object whenever this information should be logged. In our experiments related to logging the diagnostics information the HFT-version was better by 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 case we needed to modify 6 files and 13 functions. 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 advices. No modifications of the functions or lines of code is required and only one aspect was added. If we consider only exception logging, then the non-HFT version will lose by the "lines of code" metric as well.

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

TABLE II. THE HFT VERSION

Changes	Lines of code			Functions / Advices			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 III. THE NON-HFT VERSION

Changes	Lines of code			Functions / Advices			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

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. 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, fewer new and modified functions.

Regarding the combined analysis of *lines of code affected, functions/advices 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 OO approach.

Some metrics such as cohesion, coupling, separation of concerns and changeability do not directly depend on affected LoC 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 cohesion and decreases coupling.

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 almost is 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 will 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 for better cohesion and separation of crosscutting concerns.

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 risks 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 suitable action. Here is a risk of locks 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 deadlocks and bottlenecks.

The evaluation showed that for the set of the changes used the maintainability of the HFT version most of the time is better than maintainability of the non-HFT version. In the remaining cases, the HFT version required approximately the same efforts as the non-HFT version based on OOP approach. However, there are not any modifications for which the HFT version is worse than the non-HFT version. It is very important that the HFT version provides much better cohesion and separation of concerns than the non-HFT version.

VI. CONCLUSION

During their lifetime software systems require various maintenance works to add new features and fix the discovered bugs. These modifications are related to functional and non-functional features. In this study, we presented an experimental evaluation of Holistic Fault Tolerance architecture based on the typical changes of FT-related code. According to the experimental results there are clear benefits of using the HFT architecture implemented with AOP. In the most cases, the HFT version is better for maintainability. Thus, the HFT architecture can be applied to improve maintainability of FT mechanisms in the application.

Currently we are working on the techniques that will support the modelling of the HFT architecture for various computer systems. The modelling is closely interconnected with HFT efficiency evaluation, since the model will assist in choosing of the components that interact with the HFT elements. In addition, the model will allow the developer to adjust the HFT elements and their link with application components to achieve efficient operation of the application.

As a future work we consider developing adaptive holistic fault tolerance that can self-tune by introducing or switching-off the HFT agents and reconfiguring the HFT controller depending on current system loading. This will help to ensure the scalability of the HFT approach regardless of the system size.

This study focuses on the experimental evaluation of fault tolerance maintainability. Unfortunately, to the best of our knowledge there is not much work in the area. Very often the designers of new fault tolerance techniques rely on intuition and experience when they claim the maintainability of these solutions. It is our hope that this work will be helpful and useful in demonstrating how this evaluation could be conducted.

REFERENCES

[1] R. Gensh, A. Romanovsky, A. Yakovlev, “On structuring holistic fault tolerance,” in *Proceedings of the 15th International Conference on Modularity*. ACM, New York, USA, 2016, pp. 130-133.

[2] R. Gensh, A. Garcia, F. Xia, A. Rafiev, A. Romanovsky, A. Yakovlev, “Architecting Holistic Fault Tolerance,” in *Proceedings of the 18th International Symposium on High Assurance Systems Engineering*, Singapore, 2017, pp. 5-8.

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

[4] A. Avizienis, J. C. Laprie, B. Randell and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," in *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11-33, Jan.-March 2004.

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

[6] R. C. Martin, "Agile Software Development, Principles, Patterns, and Practices," Prentice Hall, 2003, p. 95–98.

[7] B. Meyer, Object-oriented Software Construction, Upper Saddle River, NJ, USA: Prentice Hall, 1988.

[8] R. C. Martin, "The Open-Closed Principle," *C++ Report*, vol. 8, no. 1, pp. 37-43, 1996.

[9] W. P. Stevens, G. J. Myers and L. L. Constantine, "Structured design," in *IBM Systems Journal*, vol. 13, no. 2, pp. 115-139, 1974

[10] E. W. Dijkstra, "On the Role of Scientific Thought," in *Selected Writings on Computing: A personal Perspective*, New York, NY, Springer New York, 1982, pp. 60-66.

[11] Microsoft Patterns & Practices Team. (2009). *NET Application Architecture Guide, 2nd Edition*. Microsoft Press.

[12] 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.

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

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

[15] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," in *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476-493, Jun 1994.

[16] M. Ceccato, P. Tonella, "Measuring the Effects of Software Aspectization,", in *Proceedings of the 1st Workshop on Aspect Reverse Engineering*, 2004.

[17] N. Cacho, "Supporting Maintainable Exception Handling with Explicit Exception Channels," PhD thesis, Lancaster University, 2009.

[18] F. C. Filho, A. Garcia, C. M. F. Rubira, "Extracting Error Handling to Aspects: A Cookbook," *2007 IEEE International Conference on Software Maintenance*, Paris, 2007, pp. 134-143.

[19] S. Karol, N. A. Rink, B. Gyapjas, J. Castrillon. 2016. "Fault tolerance with aspects: a feasibility study," in *Proceedings of the 15th International Conference on Modularity* (MODULARITY 2016). ACM, New York, NY, USA, 66-69.

[20] R. Alexandersson, P. Öhman, J. Karlsson, "Aspect-Oriented Implementation of Fault Tolerance: An Assessment of Overhead," in *Computer Safety, Reliability, and Security*, Springer Berlin Heidelberg, 2010, pp. 466-479.

[21] N. Cacho, F. Dantas, A. Garcia, F. Castor, "Exception Flows Made Explicit: An Exploratory Study," In *Proceedings of the 2009 XXIII Brazilian Symposium on Software Engineering* (SBES '09), Fortaleza, Ceara, 2009, pp. 43-53.

[22] G. Bradski, The OpenCV Library, *Dr. Dobb's Journal of Software Tools*, 2000

[23] R. Smith, "An Overview of the Tesseract OCR Engine," *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, Parana, 2007, pp. 629-633.

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