TUCS

Kashif Javed

Model-Driven Development and Verification of Fault Tolerant Systems

Turku Centre *for* Computer Science

TUCS Dissertations
No 223, May 2017

Model-Driven Development and Verification of Fault-Tolerant Systems

Kashif Javed

To be presented, with the permission of the Faculty of Science and Engineering of the Åbo Akademi University, for public criticism in Auditorium XX on May 24th, 2017, at 12 noon.

Turku Centre for Computer Science
Åbo Akademi University
Faculty of Science and Engineering
Agora, Universitetsbacken, 20520 Åbo
Finland

2017

Supervised by

Docent Elena Troubitsyna
Faculty of Science and Engineering
Åbo Akademi University
Vattenborgsvägen 5, 20500 Turku
Finland

Reviewed by

Professor Anatoliy Gorbenko
Department of Computer Systems and Networks
National Aerospace University (KhAI)
Kharkov, Ukraine

Professor Amund Skavhaug
Faculty of Engineering Science and Technology
Norwegian University of Science and Technology (NTNU)
Trondheim, Norway

Opponent

Professor Anatoliy Gorbenko
Department of Computer Systems and Networks
National Aerospace University (KhAI)
Kharkov, Ukraine

ISBN 978-952-12-3553-5
ISSN 1239-1883

Dedicated to my dearest parents who have supported me throughout my education

“Seek knowledge from the cradle to the grave."

Prophet Muhammad (Peace be upon him)

Abstract

Dependability is an ability of a computer-based system to deliver services that can be justifiably trusted. There is a wide range of computer-based systems that provide services that are critical for our society, e.g., nuclear power plants, transportation, healthcare, etc. Ensuring dependability of such systems constitutes an important engineering goal.

Dependability is an integrated notion that encompasses various system characteristics including reliability, safety, availability, etc. For a wide class of systems, the main engineering goal is to ensure a high degree of reliability – a probability of a system functioning correctly over a given period under a given set of operating conditions. Since the occurrence of faults can disrupt correct system behavior, to achieve the required reliability, we need to employ fault tolerance techniques.

The main goal of fault tolerance is to ensure that the system can deliver its services despite the occurrence of faults. Fault tolerance typically introduces some form of architectural or computational redundancy and hence, increases the complexity of the system design. Therefore, to ensure a correct implementation of fault tolerance mechanisms, we should develop the techniques that facilitate a structured analysis of system failure modes, systematic design of error recovery and reconfiguration procedures, as well as bolster system verification and validation.

Model-driven engineering allows the designers to cope with system complexity and analyze system behavior at different levels of abstraction. In our thesis, we aim at studying how to represent various static and dynamic aspects of fault tolerance in the model-driven development.

Often behavior of complex fault-tolerant systems is structured using the notion of operational modes – mutually exclusive sets of the system behavior. As a reaction on faults as well as different internal and external conditions, the system switches between its operational modes. The design of mode transition logic in distributed fault tolerant systems is a challenging and error-prone task. On the one hand, we need to ensure that all components of the system are put in the states required by a certain mode. On the other hand, we should verify that the components maintain these states while the system is stable, i.e., before the conditions for triggering a transition to another mode are reached. To facilitate the design of complex mode-rich fault tolerant systems, in our thesis, we demonstrate how to model distributed systems with centralized and distributed mode management as well as verify their mode transition logic. We validate the proposed approach in the aerospace domain.

To guarantee that the reconfiguration performed in response to the changed operating conditions achieves the required goal, fault-tolerant systems should incorporate the appropriate monitoring capabilities and rely on a set of explicitly defined rules for triggering adaptation. Usually, these rules establish a connection between globally observed system properties and behavior of system components, i.e., span over several architectural layers. To facilitate the development of such complex systems, in our work, we propose the generic patterns for architecting adaptive fault tolerant systems in a layered hierarchical manner. We demonstrate how the proposed patterns can be utilized in the context of data intensive fault tolerant systems.

Development and verification of fault-tolerant distributed systems also require the use of advanced verification technologies as well as techniques for supporting a disciplined systematic analysis of system failure modes. In our thesis, we demonstrate how to perform formal verification of a fault tolerant routing protocol and systematically identify system failure modes and recovery procedures using Failure Modes and Effect Analysis approach. While reliance on formal techniques increases confidence in the correctness of the implementation of the fault tolerance mechanisms, the use of a systematic inductive technique for identifying failure modes improves the completeness of the analysis.

The research work performed in our thesis aims at creating a support for explicit integration of fault tolerance consideration into the model-driven system development. We aim at creating a potentially industry-relevant approach and hence, utilize the modeling and verification techniques that are used in current industrial practice. Moreover, we validate our approach in a number of case studies from different domains.

Sammanfattning

Tillförlitlighet är ett datorbaserat systems förmåga att kunna leverera tjänster som med goda argument är pålitliga. Det finns ett brett spektrum av datorbaserade system som tillhandahåller tjänster som är avgörande för vårt samhälle, t.ex. kärnkraftverk, transport, hälso- och sjukvård osv. Att säkerställa pålitligheten för sådana system är ett viktigt tekniskt mål.

Tillförlitlighet är ett integrerat begrepp som omfattar olika systemegenskaper, inklusive funktionssäkerhet, säkerhet, tillgänglighet etc. För en omfattande klass av system är det huvudsakliga tekniska målet att säkerställa en hög grad av tillförlitlighet - en sannolikhet för att ett system fungerar korrekt under en viss period med en given mängd driftsförhållanden. Eftersom fel som uppstår kan störa ett korrekt systembeteende, måste vi använda feltoleranstekniker för att uppnå den tillförlitlighet som krävs.

Huvudmålet med feltolerans är att säkerställa att systemet kan leverera sina tjänster trots att fel uppstått. Feltolerans introducerar typiskt någon form av arkitektonisk eller beräkningsmässig redundans och ökar därigenom systemdesignens komplexitet. För att säkerställa ett korrekt genomförande av feltoleransmekanismer bör vi utveckla de tekniker som underlättar en strukturerad analys av systemfelslägen, systematisk utformning av felhantering och procedurer för omkonfiguration, samt förstärkning av systemets verifiering och validering.

Modelldriven teknik möjliggör hantering av systemkomplexitet och analys av systembeteendet på olika abstraktionsnivåer. Avhandling strävar efter att studera hur man representerar olika statiska och dynamiska aspekter av feltolerans i modelldriven utveckling.

Ofta är beteendet hos komplexa feltoleranta system strukturerat enligt operativa lägen - ömsesidigt uteslutande uppsättningar av systembeteendet. Som en reaktion på fel såväl som olika interna och externa förhållanden växlar systemet mellan sina driftlägen. Utformningen av logik för lägesövergång i distribuerade feltoleranta system är en utmanande och felbenägen uppgift. Å ena sidan måste vi se till att alla komponenter i systemet sätts i de tillstånd som krävs av ett visst läge. Å andra sidan bör vi verifiera att komponenterna upprätthåller dessa tillstånd medan systemet är stabilt, dvs. innan förutsättningarna för en övergång till ett annat läge uppnås. För att underlätta utformningen av komplexa feltoleranta system med många tillstånd, visar vi i vår avhandling hur man modellerar distribuerade system med centraliserad och distribuerad lägeshantering samt verifierar deras logik för lägesövergång. Vi validerar det föreslagna tillvägagångssättet inom rymdindustrin.

För att säkerställa att omkonfigurationen som utförts som svar på de ändrade driftsförhållandena uppnår det önskade målet, bör feltoleranta system inkludera lämplig övervakningsförmåga och förlita sig på en uppsättning uttryckligen definierade regler för start av anpassning. Vanligtvis etablerar dessa regler en koppling mellan globalt observerade systemegenskaper och beteenden hos systemkomponenter, dvs. de spänner över flera arkitektoniska skikt. För att underlätta utvecklingen av sådana komplexa system, föreslår vi i vårt arbete generiska mönster för konstruktion av adaptiva feltoleranta system på ett hierarkiskt sätt. Vi visar hur de föreslagna mönstren kan utnyttjas i samband med dataintensiva feltoleranta system.

Utveckling och verifiering av feltoleranta distribuerade system kräver också användning av avancerad verifieringsteknik samt tekniker för att stödja en systematisk analys av systemets fellägen. I vår avhandling visar vi hur man utför formell verifiering av ett feltolerant ruttningsprotokoll och systematiskt identifierar systemets fellägen och återställningsförfaranden med hjälp av FMEA (Failure Modes and Effect Analysis). Medan formella tekniker ökar konfidensen för korrekt implementering av feltoleransmekanismerna, förbättrar användningen av en systematisk induktiv teknik för identifiering av fellägen analysens fullständighet.

Forskningsarbetet i avhandlingen strävar efter att skapa stöd för en explicit integration av feltolerans i modelldriven systemutveckling. Vi strävar efter att skapa ett tillvägagångssätt som är relevant inom industrin och därför utnyttjar vi de modellerings- och verifieringsmetoder som används i nuvarande industripraxis. Dessutom validerar vi vårt tillvägagångssätt i ett antal fallstudier från olika domäner.

Acknowledgements

First of all, I thank Almighty Allah for giving me the courage and strength to pursue my PhD research work at Åbo Akademi University, Turku, Finland. This thesis would not have been possible without the financial support of National University of Sciences and Technology, Islamabad, Pakistan.

I would like to express my gratitude to my supervisor, Associate Professor Elena Troubitsyna, for her professional guidance, full-time supervision, close monitoring and timely feedback, which has enabled me to complete this thesis. I wholeheartedly appreciate her continued support, great encouragement, valuable advice, and friendly atmosphere during the whole PhD research work. I am indebted to her for ensuring the provision of conducive and productive research environment in the Distributed Systems Laboratory of the Department of Information Technology and Turku Centre for Computer Science (TUCS). I am very grateful to the administrative and technical staff of the department and TUCS for providing full assistance during my research work.

I would like to thank my reviewers, Professor Anatoliy Gorbenko from National Aerospace University and Professor Amund Skavhaug from Norwegian University of Science and Technology, for sparing their valuable time to read the thesis and giving extremely useful suggestions for further refining and enhancing the quality of the thesis work. I am very thankful to Professor Anatoliy Gorbenko for accepting to be the opponent in the public defense of my thesis. I owe my special thanks to Professor Anatoliy Gorbenko for his precious time and efforts to act as a proof reader of my thesis. I would also like to thank Johan and Jonathan for translating the abstract of the thesis into Swedish.

I would like to extend my special appreciation to Christel Engblom who always helped me to conduct the experimental work by making necessary arrangements in the laboratory. I am grateful to Magnus Dahlvik for providing requisite computer support. I would also like to thank from the core of my heart Hans Bang Stiftelsen, Ulla Tuominen and Åbo Akademi University for awarding me the research grants during my PhD work.

I enjoyed the study and research environment provided by the Department of Information Technology and TUCS. I would like to express my special thanks to my colleagues, working in the Distributed Systems Laboratory and ICT building, for giving me technical support and providing cooperation during the tough hours of research work.

I am very grateful to my parents, Professor Muhammad Younus Javed and Manzoor Akhtar, whose prayers and patronage have played a pivotal role in the completion of this work. I would also like to extend my sincere appreciation to Dr. Qaisar Javed, Shumaila Younus and Munazza Younus who have always been a great source of inspiration and encouragement for me.

Last but not the least; I owe countless thanks to my wife, Asifa Kashif, and my son, Muhammad Umer Javed, for standing beside me throughout this research journey. Their lovely attitude, persistent support, thorough understanding, humorous behavior and sincere prayers have greatly helped me to complete this thesis within the scheduled timeline.

Åbo, May 2017
Kashif Javed

List of Original Publications

I Kashif Javed, Asifa Kashif, and Elena Troubitsyna. Implementation of SPIN Model Checker for Formal Verification of Distance Vector Routing Protocol. In *International Journal of Computer Science and Information Security* (IJCSIS), Vol 8, No 3, pp. 1-6, ISSN 1947-5500, June 2010, USA.

II Kashif Javed, Elena Troubitsyna. Designing a Fault-Tolerant Satellite System in SystemC. In *Proceedings of the Seventh International Conference on Systems (ICONS 2012)*, pp. 49–54, IEEE Computer Press, March 2012, Saint Gilles, Reunion Island.

III Kashif Javed and Elena Troubitsyna. Modelling a Fault-Tolerant Distributed Satellite System. In *Proceeding of the International Conference Advanced Collaborative Networks, Systems and Applications (COLLA 2012),* pp. 35-41, June 2012, Venice, Italy.

IV Kashif Javed and E. Troubitsyna, A Case Study in Modelling a Fault-Tolerant Satellite System Implementing Dynamic Reconfiguration via Handshake. In *Proceedings of the Seventh International Conference on Software Engineering Advances (ICSEA2012)*, pp. 44-49, November 2012, Lisbon, Portugal.

V Elena Troubitsyna and Kashif Javed. Towards Systematic Design of Adaptive Fault-Tolerant Systems. In *Proceedings of the Sixth International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE 2014)*, pp. 15-21, May 2014, Venice, Italy.

VI Elena Troubitsyna and Kashif Javed. A Structured Approach to Architecting Fault-Tolerant Services. In *Proceedings of the Ninth International Conference on Internet and Web Applications and Services (ICIW 2014)*, pp. 99-104, July 2014, Paris, France.

Contents

Research Summary 1
1 Motivation and Research Objectives 3
2 Dependability and Fault Tolerance 7
2.1 Dependability Taxonomy 7
2.2 Fault Tolerance 10
3 Design and Verification Methods 17
3.1 UML 17
3.2 SystemC 22
3.3 Model Checking Fault-Tolerant Systems 27
3.3.1 Model Checking: Introductory Background 27
3.3.2 SPIN PROMELA 30
4 Fault Tolerance in Mode-Rich System 35
4.1 Modes 35
4.2 Mode-Rich Fault-Tolerant Systems: Centralized Mode Management 37
4.3 Mode-Rich Fault-Tolerant Systems: Distributed Mode Management 44
5 Fault Tolerance in Service-Oriented Distributed Systems 51
5.1 Fault Tolerance in Service Oriented Architectures 51
5.2 Adaptive Fault-Tolerant Systems 55
6 Summary of the Original Publications 61
7 Related Work 65
7.1 Modes and Fault Tolerance 65
7.2 Model-Driven Engineering and Verification of Fault-Tolerant Systems. 67
7.3 Centralized Mode Management in Fault-Tolerant systems 68
7.4 Fault Tolerance in Distributed Systems 69
8 Conclusion and Future Work 73
8.1 Conclusions 73
8.2 Future Work 75
Original Publications 87

List of Figures

Figure 2.1: Dependability Taxonomy 9
Figure 2.2: Fault Tolerance Techniques... 10
Figure 2.3: Triple Modular Redundancy.. 14
Figure 2.4: Dynamic redundancy: standby spare... 15
Figure 2.5: Dynamic redundancy: duplication pattern....................................... 15
Figure 3.1: Example of a Use Case Diagram... 18
Figure 3.2: Example of a Class Diagram 19
Figure 3.3: Example of a Sequence Diagram... 20
Figure 3.4: Examples of State Diagram (a) with explicit representation of failure (b) with explicit representation of failure and error recovery 21
Figure 3.5: Definition of simple SystemC module 23
Figure 3.6: SystemC template for creating ports 24
Figure 3.7: Example of a module with a method process declaration 25
Figure 3.8: SystemC Implementation of Fault Tolerant Service Pattern 27
Figure 3.9: Model Checking Process 30
Figure 3.10: Promela Implementation of Fault-Tolerant Service 32
Figure 3.11: Example of SPIN output.. 33
Figure 4.1: Mode Logic of Fault-Tolerant System: a General Principle 37
Figure 4.2: Generic Architecture of Mode Rich Systems 38
Figure 4.3: Centralized AOCS Architecture 39
Figure 4.4: Centralized Mode-Rich Fault-Tolerant System: Example of GPS unit 40
Figure 4.5: A general pattern for defining Mode Transition in SystemC (GPS instance) 42
Figure 4.6: Example of Mode Transition Verification....................................... 43
Figure 4.7: State Diagram of D-AOCS (GPS example) 46
Figure 4.8: Distributed Mode Management (GPS example) 48
Figure 4.9: Distributed Mode Management: Verification of Handshake Procedure (Mode D example) 49
Figure 5.1: Architectural Patterns for Introducing Redundancy 52
Figure 5.2: Service Execution Flow... 53
Figure 5.3: Generic Form of FMEA Table 54
Figure 5.4: Example of Failure Analysis using FMEA...................................... 55
Figure 5.5: Architecture of an Adaptive Fault-Tolerant System........................ 57
Figure 5.6: Transitions between Adaptation Modes 58

Part I

Research Summary

1 Motivation and Research Objectives

Dependability is an ability of a computer-based system to deliver services that can be justifiably trusted. There is a wide range of computer-based systems that provide services that are critical for our society, e.g., nuclear power plants, transportation, healthcare, etc. Ensuring dependability of such systems constitutes an important engineering goal.

Dependability is an integrated notion that encompasses various system characteristics including reliability, safety, availability, etc. For a wide class of systems, the main engineering goal is to ensure a high degree of reliability – a probability of a system functioning correctly over a given period under a given set of operating conditions. Since the occurrence of faults can disrupt correct system behavior, to achieve the required reliability, we need to employ fault tolerance techniques.

The main goal of fault tolerance is to ensure that the system can deliver its services despite the occurrence of faults. Fault tolerance typically introduces some form of architectural or computational redundancy and hence, increases the complexity of the system design. Therefore, to ensure a correct implementation of fault tolerance mechanisms, we should develop the techniques that facilitate a structured analysis of system failure modes, systematic design of error recovery and reconfiguration procedures, as well as bolster verification and validation of complex fault-tolerant systems.

This goal can be achieved by employing a model-driven development approach. Reliance on abstraction and rigorous mathematical analysis provides us with a powerful support while designing complex fault tolerant systems. However, despite the popularity of model-driven engineering, there is still a lack of approaches supporting an *explicit* modeling of fault tolerance aspects of system behavior. Our first research question

> *Research question 1. How can we explicitly represent fault tolerance in the model-driven system development?*

aims at addressing this issue.

Often the behavior of complex fault-tolerant systems is structured using the notion of operational modes – mutually exclusive sets of the system behavior. As a reaction on faults as well as different internal and external conditions, the system switches between its operational modes. The design of mode transition logic in distributed fault tolerant systems is a challenging and error-prone task. On the one hand, we need to ensure that all components of the system are put in the states required by a certain mode. On the other hand, we should verify that the components maintain these states while the system is

stable, i.e., before the conditions for triggering a transition to another mode are reached.
To address this challenge, we formulate the second research question:

> *Research question 2: How can we facilitate design and verification of complex mode-rich fault tolerant systems with centralized and distributed mode management?*

Development and verification of fault-tolerant distributed systems also require the use of advanced verification technologies as well as techniques for supporting a disciplined systematic analysis of system failure modes. Moreover, to guarantee that the reconfiguration performed in response to the changed operating conditions achieves the required goals, fault tolerant systems should incorporate the appropriate monitoring and adaptation capabilities. Our third research question:

> *Research question 3. How can we facilitate systematic analysis of system failure modes, structure system architecture to support adaptability and verify fault tolerance capabilities?*

aims at studying these problems.

In our thesis, we aim at finding potentially industry-relevant solutions addressing the identified research questions. Therefore, we will utilize the modeling and verification techniques that are used in the current industrial practice. Moreover, we will validate the proposed solutions in a number of case studies from different industrial domains.

Organization of the Thesis. This thesis consists of two parts. The overview of the research work reported in this thesis is included in Part I. Part II contains the original research publications.

Part I is organized as follows. Chapter 2 discusses the dependability concept and pays special attention to fault tolerance as one of the essential dependability attributes.

Chapter 3 overviews the development methodologies, which have been used in the thesis. In this section, we discuss UML, SystemC, SPIN model checker and the associated modeling language PROMELA. We illustrate a representation of different aspects of fault tolerance in the corresponding frameworks.

Chapter 4 focuses on the concept of modes and main issues in designing fault-tolerant mode-rich systems. We identify the problems and solutions in implementing distributed systems with the centralized as well as distributed mode management. We use a case study from the aerospace domain to illustrate the principles of achieving fault tolerance via mode transitions. In particular, we demonstrate how to address the issue of ensuring correctness of mode logic. Moreover, we explain how to use the handshake protocol to synchronizing mode transitions in the distributed mode management.

Chapter 5 focuses on studying architectural aspects of fault tolerance. Firstly, we discuss how to modify and apply Failure Mode and Effect Analysis (FMEA) technique to systematically identify system failure modes and define architectural patterns for error masking or recovery. Then we outline the generic principles of architecting adaptable fault tolerant systems.

A detailed description of the published research papers is given in Chapter 6. The overview of the related work is presented in Chapter 7. Chapter 8 summarizes the main contributions of the research work carried out in this thesis, discusses its strengths and limitations, as well as outlines the future research directions.

2 Dependability and Fault Tolerance

In this chapter, we give an overview of the dependability concept and in particular, focus on the fault tolerance aspect.

2.1 Dependability Taxonomy

Dependability of a computing system is the ability to deliver service that can justifiably be trusted. The notion of dependability was introduced by Laprie [1] and further refined by Avizienis et al [2]. Nowadays complex computer-based systems are embedded in the infrastructures supporting the majority of critical services provided to our society. Therefore, dependability has become the concern of the highest priority in the development and operation of modern computer-based systems.

The variety of computing systems on whose services we need to place our reliance is broad – it ranges from satellite constellations, airplanes, nuclear power plants, or databases containing sensitive health records. Correspondingly, different characteristics of system behavior become the main priority in their development. For instance, for the satellite systems we need to guarantee a high degree of reliability, i.e., to ensure that despite an occurrence of faults and other environmental disturbances, the system can continuously operate for a certain period. For the airplanes and nuclear power plants, we should ensure that the system is safe, i.e., the likelihood of occurrence of hazardous failures is very low. Finally, for the databases containing sensitive data, we have to ensure a high degree of security, i.e., the absence of unauthorized access or data alternations. The concept of dependability provides us with a unified framework that allows us to address such diverse concerns within a single conceptual framework. It consists of three parts: dependability attributes, threats to dependability and means for achieving dependability. Figure 2.1 presents the dependability taxonomy [1].

The key attributes of dependability are given below:

- *Availability*: the ability of the system to provide a correct service at any given moment in time.
- *Reliability:* the ability of the system to provide a service under a given set of operating conditions over a specific time interval.

- *Safety:* the ability of the system to provide a service under the given conditions without jeopardizing its environment and users.
- *Maintainability:* the ability of the system to undergo repairs and modifications.
- *Integrity:* the ability of the system to prevent improper state alterations.
- *Confidentiality:* the absence of unauthorized disclosure of information.

The *threats* to dependability, often referred to as the impairments to dependability may introduce the unwanted alterations in the service provisioning. Dependability is impaired by the occurrence of faults, errors, and failures.

- *Fault*: a defect within the system. Faults can be generated due to internal factors (e.g., software coding mistakes, memory bit "stuck") and external factors (e.g., component defects or human mistakes). Faults may result in errors.
- *Error:* a deviation from the required operation of the system or subsystem. Errors are the effect of faults and may lead to subsequent system failure.
- *Failure:* a deviation in provisioning a required service to the system user. Failures are the effects of errors.

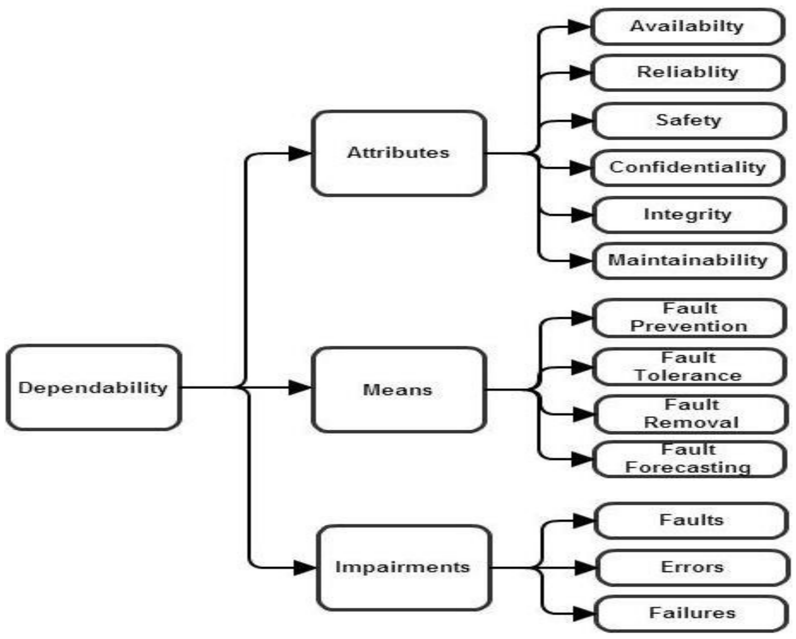


Figure 2.1: Dependability Taxonomy

The *means* for dependability are the techniques used to facilitate the development of dependable systems. The dependability means can be classified into four categories: fault prevention, fault removal, fault forecasting and fault tolerance.

- *Fault prevention:* the techniques aiming at reducing the likelihood of an introduction of faults during the process of system development.
- *Fault removal:* the techniques that facilitate identifying and removing the faults during the development stage as well as during the operational life of a system.
- *Fault forecasting:* the techniques that are applied to predict fault occurrence and evaluate their possible consequences on the system behavior.
- *Fault tolerance:* the techniques that ensure that the system can continue to deliver its services even in the presence of faults.

Since our thesis focuses on studying methods for achieving fault tolerance, below we give a detailed overview of the fault tolerance concept.

2.2 Fault Tolerance

The main aim of fault tolerance [3] is to ensure that the system continues to provide its services even in the presence of faults. Typically, fault occurrence leads to a certain service degradation. However, it is important to ensure that the system behaves in a predictable deterministic way even in the presence of faults.

The main techniques to achieve fault tolerance are error processing and fault treatment, as shown in Figure 2.2.

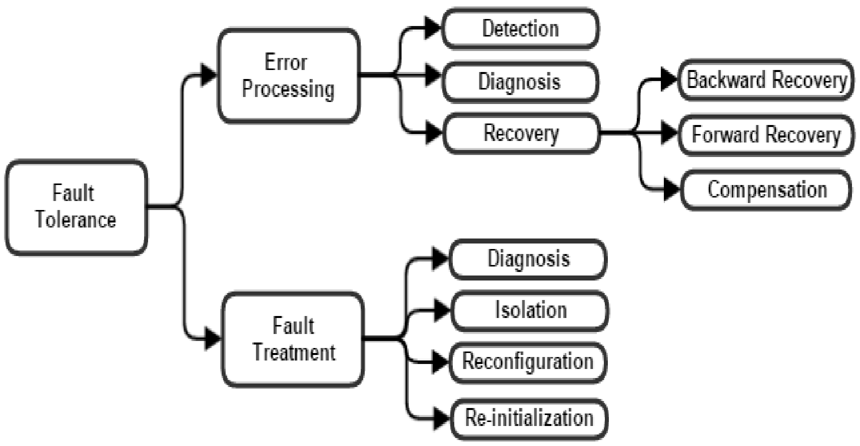


Figure 2.2: Fault Tolerance Techniques

Error processing: Error processing comprises the measures applied while the system is operational. The purpose of error processing is to eliminate an error from the computational state and preclude failure occurrence. Error processing is usually implemented in three steps: error detection, error diagnosis, and error recovery [15].

- *Error detection:* determines the presence of error.
- *Error diagnosis:* evaluates the amount of damage caused by the detected error.
- *Error recovery:* aims at replacing an erroneous system state with the error-free state.

  There are three types of error recovery methods: backward recovery, forward recovery and compensation.

- *Backward recovery:* tries to return the system to some previous error-free state. Typically, backward recovery is implemented by checkpointing, i.e., periodically, during the normal system operation, the state of the system is stored in the memory. In the case of a failure, the system retrieves the information about the error-free state from memory and resumes its functioning from this state.
- *Forward recovery:* upon detection of an error, the system makes a transition to a new error-free state from which it continues to operate. Exception handling is a typical example of forward error recovery.
- *Compensation:* can be used when the erroneous state contains enough redundancy to enable its transformation to an error-free state. Compensations are often used in complex transactions.

Fault treatment: aims at preventing faults from being activated again. Fault treatment is usually performed while the system is not operational, i.e., during the scheduled maintenance. Fault treatment comprises four steps: diagnosis, isolation, reconfiguration, and re-initialization.

- *Diagnosis* determines the causes of errors and focuses on localizing a fault and determining its nature.
- *Isolation* prevents faulty components from being used or activated in further system operations.
- *Reconfiguration* replaces the faulty components with the fault-free ones to provide an acceptable but possibly degraded service.
- *Re-initialization* is an update performed after the new configuration has taken place.

To implement fault tolerance, it is important to understand the types of faults that might occur in the system. Faults can be characterized as nature, duration or extent.

- *Nature:* We distinguish between random and systematic faults. The random faults are associated with hardware components. For the components that are the subjects of random faults, we can use the statistical analysis and estimate various characteristics, e.g., such as mean time between failures. Systematic faults are typically associated with the design errors, e.g., mistakes in the system specification or implementation. Since an error will occur each time the erroneous state is reached, statistical methods cannot be applied to the systematic faults. Systematic faults should be prevented by the use of fault avoidance and fault removal techniques.

- *Duration:* Faults can be classified regarding their duration into permanent, transient, and intermittent faults.
    - Permanent faults: once they have occurred, they remain in the system during its entire operational life, if no corrective actions are performed.
    - *Transient* faults can appear and then disappear after a short time.
    - *Intermittent faults*: can appear, disappear and then reappear at a later time.
- *Extent:* Faults can be categorized according to their effect on the system as localized and global ones.

    - *Localized* faults affect only a single hardware or software module.
    - *Global* faults permeate throughout the system.

Hardware faults can be due to either random component failure or mistakes in the design. The faults can be permanent, transient, or intermittent and can have a global or local extent. Software faults are systematic. They occur due to mistakes in the design of the system. These faults can have an unlimited number of forms, e.g., coding faults, stack overflows or underflows, logical errors in calculations, use of uninitialized variables, etc.

To detect and recover from faults, we have to introduce some form of redundancy into the system design. *Redundancy* can be defined as the use of resources or components that would not have been needed if the systems were fault free. Next, we overview different forms of redundancy.

Hardware redundancy is defined as the use of additional hardware to detect or tolerate faults. Static, dynamic and hybrid redundancy are the three basic forms of hardware redundancy.

- *Static redundancy* allows the designers to implement fault masking, i.e., it allows the system to nullify the effect of fault occurrence.
- *Dynamic redundancy:* this form of redundancy allows the system to detect the faults and then perform reconfiguration to deactivate faulty components.
- *Hybrid redundancy:* This form of redundancy combines static and dynamic techniques. Fault masking is used to prevent the error propagation within the system, whereas fault detection and reconfiguration are used to remove faulty components from the system.

Information redundancy: The use of additional information that is required to implement a given function for the purpose of detecting or tolerating faults. The use of parity bits, checksums, and error detecting or correcting codes are the examples of this kind of redundancy.

Temporal redundancy: The use of additional time that is required to implement a given function for the purpose of detecting or tolerating faults. Temporal redundancy is used to tolerate transient faults, e.g., by repeating the failed computation.

Software redundancy can be defined as the use of additional software for the purpose of detecting or tolerating faults. Designing software that controls fault tolerance-related functionality is one of the main topics of our research work.

In this thesis, we consider static and dynamic forms of redundancy. The static redundancy relies on a voting mechanism that compares the outputs of some redundant modules and produces a majority view. This mechanism allows the system to mask an occurrence of faults. In practice, static redundancy is often implemented as the triple modular redundancy arrangement, as shown in

Figure **2.3**. The arrangement consists of three identical modules and one voting element. If the system is fault free, then each module produces the same output. Any difference between the outputs represents a failure in the module. A voting element nullifies the effect of any single failure by comparing the outputs and generating the output corresponding to the majority view. Therefore, the system can tolerate the failure of any single module.

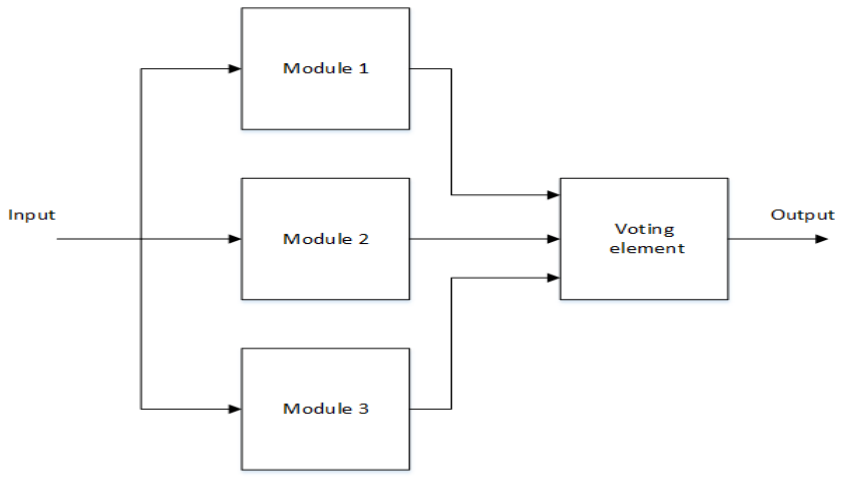


Figure 2.3: Triple Modular Redundancy

Dynamic forms of redundancy employ fault detection instead of fault masking approach. These systems attempt to detect faults and then to reconfigure to continue an error-free function. The effectiveness of the fault detection process determines the success of the dynamic redundancy. This approach minimizes the required number of redundant components because only two modules are required to cope with a single fault and three modules to cater for two faulty units. The examples of dynamic redundancy include a standby spare and a duplication pattern, as shown in

Figure **2.4** and Figure 2.5 respectively.

- **Standby spare:** In the standby spare arrangement, one module is in the operational state, while the other is in the standby state. A fault detection system is used to detect the faults and control the switch. In case the system is fault free, the switch generates the output corresponding to the output from the single operational module. When a fault is detected, the switch will reconfigure the system and generate the output corresponding to the output produced by the standby module.
- **Duplication pattern:** In the duplication pattern, the same input signal is fed into two identical modules. The comparator compares the outputs from both modules and generates a failure detection signal in case a discrepancy is detected. One of the module’s output is then passed to the next stage.

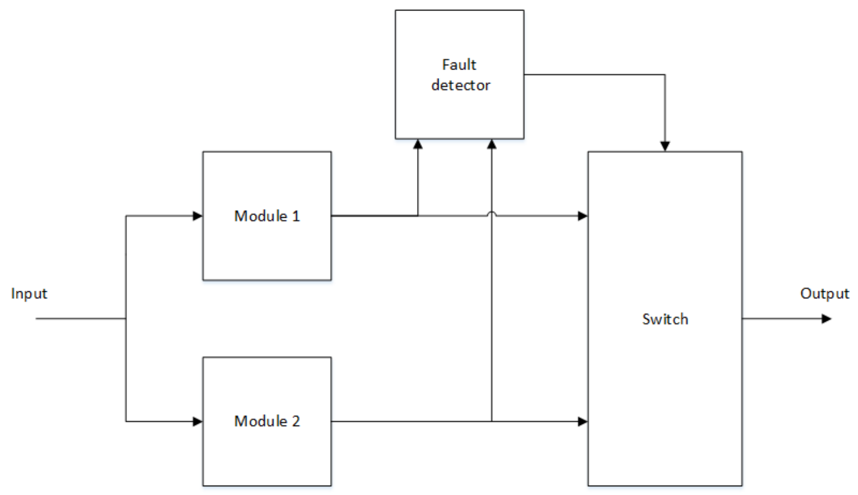


Figure 2.4: Dynamic redundancy: standby spare

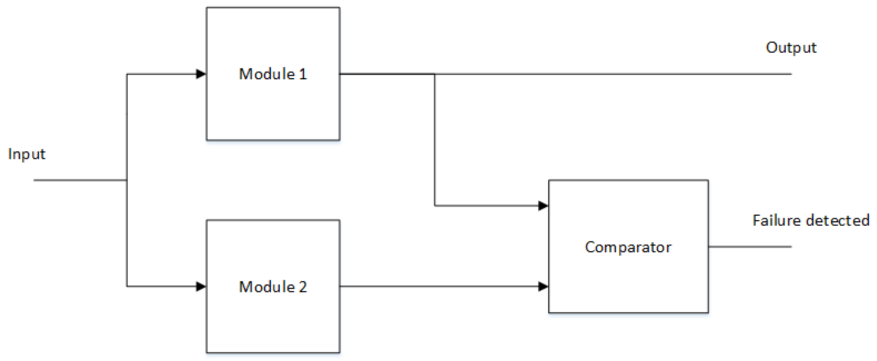


Figure 2.5: Dynamic redundancy: duplication pattern

In modern computer-based systems, fault tolerance is often implemented by software. The software is responsible for detecting faults, initiating error recovery and performing reconfiguration. To design software, which is responsible for implementing fault tolerance, we should analyze some factors including the nature of faults, possibility to introduce hardware redundancy in the system design, timing and memory constraints, and availability of information and timing redundancy, etc. Therefore, the basic concepts that we have described above constitute an important background for designing fault-tolerant systems. However, redundancy and corresponding functionality required to control it inevitably introduce additional complexity into the system design. Since complexity is commonly perceived as one of the main dependability threat, we should employ a disciplined software development process and formal reasoning to ensure functional correctness of fault-tolerant software-intensive systems.

In the next chapter, we overview the methodologies that we used in our thesis to design software that is responsible for fault tolerance assurance.

3 Design and Verification Methods

To cope with the complexity of modern software-intensive systems, we should rely on structured and rigorous approaches for software development. Such approaches allow us to build robust system architectures and ensure functional correctness of system behavior.

The range of modeling techniques is broad and spans from graphical notations to formal mathematical languages. Visual modeling frameworks are widely used in industry. However, in the safety-critical domain, formal approaches are also used to verify critical system functions. Since, in our thesis, we aim at proposing a practice-oriented approach to development and verification of fault-tolerant systems, in our work we adopt both graphical and formal modeling techniques. Next, we overview them and present various examples of modeling fault tolerant aspects.

3.1 UML

Unified Modelling Language (UML) [16, 19] is a framework for visualizing and documenting software systems. It has become de-facto industry standard graphical notation for describing software analysis and designs. UML employs object-oriented style and is independent of a specific programming language [23].

UML supports model-driven software development. It is a part in a variety of applications including complex distributed systems. The language employs a set of specific symbols to graphically represent various components and relationships between them [16, 19]. There are several automated environments that support modeling in UML. Some of them also offer a possibility to generate the program code from the defined graphical models.

The most useful UML diagrams are use case diagram, class diagram, sequence diagram, state diagram, activity diagram and component diagram. In our thesis, we mainly used a subset of them that comprises use case diagram, class diagram, state diagram and sequence diagram. We describe them next.

A *use case* represents a unit of functionality provided by the system [9]. The main goal of the use-case diagram is to support the development teams in visualizing the functional requirements of a system and defining the relationships with actors – the human users or other subsystems interacting with the modeled system. An example of a use case diagram is shown in

Figure **3.1**.

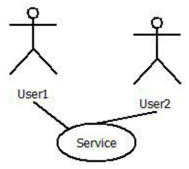


Figure 3.1: Example of a Use Case Diagram

The diagram shows two actors *User1* and *User2* that interact with the system while it provides a functionality called *Service.* Often the use case diagram represents the essential system processes and relationships among different use cases. Typically, a use-case diagram shows a group of use cases — either all use cases of the system under construction or a particular subset of use cases with related functionality.

The *class diagram* [11] shows how system entities relate to each other, i.e. it defines the static structure of the system [9]. The class diagrams are usually used to display the classes about the real-world purpose of the system, so called logical classes. The class diagrams also depict the relationships between the classes, such as inheritance or association.

In our work, we aim at making the fault tolerance aspect explicit during system modeling, design, and verification. Hence, we propose a number of generic patterns that allow the designers to represent system behavior not only in the nominal conditions but also in the presence of faults.

An example of a class diagram with an explicit representation of fault tolerance aspect is given in Figure 3.2. The diagram shows three classes called SD, SC, and ER. The classes are defined to explicitly represent the steps of fault tolerance-aware service architecture. The class SD is responsible for the communication with the user. Moreover, it orchestrates the work of the service executing component SC and error handler ER. SC implements the main functionality of the service. Since service execution might fail or succeed, the SC class contains attributes *failure* and *success*. The ER class implements error recovery functionality that again might succeed or fail, as designated by the corresponding attributes *rec_result* and *unrec_failure*. The dynamics of the class interactions is explained in the accompanying sequence diagram.

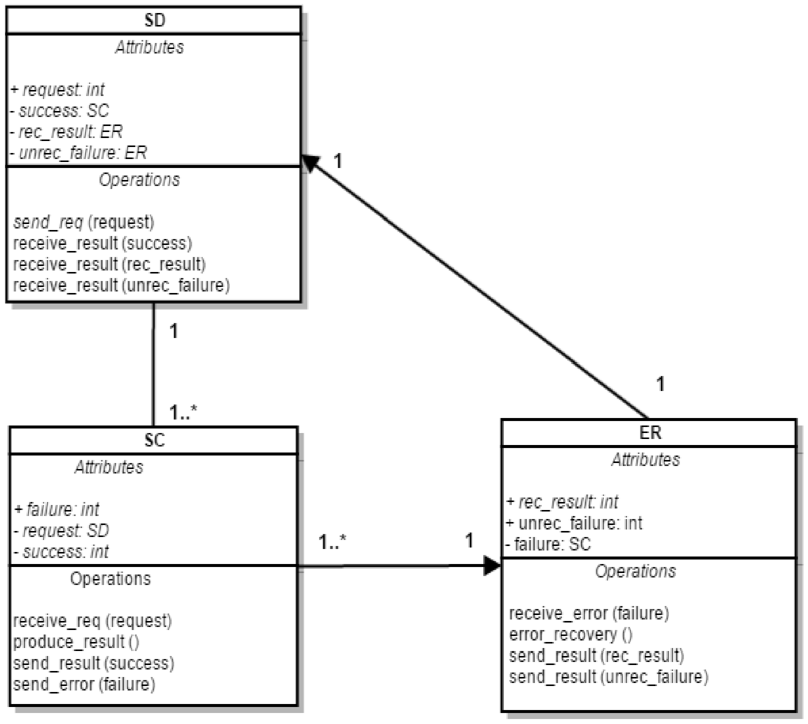


Figure 3.2: Example of a Class Diagram

A *sequence diagram* [10] shows a detailed flow of control that is typically defined for a specific use case (*Service* in our case). The sequence diagrams show the calls between the objects in a sequence. A sequence diagram has two dimensions – the vertical and horizontal. The vertical dimension shows the sequence of messages/calls in the time order that they occur. The horizontal dimension shows the object instances to which the messages are sent.

The top of the diagram contains the identities of the class instances (objects). To show that a class instance sends a message to another class instance, we draw a line with an open arrowhead pointing to the receiving class instance and place the name of the message/method above the line. The returned message is drawn with a dotted line with an arrowhead pointing back to the originating class instance. It is labeled with the return value above the dotted line.

An example of a sequence diagram is given in Figure 3.3. Here we again demonstrate how to explicitly introduce handling of faulty behavior into the service design. We define three processes – Service Director (SD), Service Component (SC) and Error Recovery (ER) as the corresponding parallel vertical lines. The interactions between the processes are carried out according to the communication between the service and its user in a time sequence.

SD plays two roles: it handles the user communication with the service and controls the service execution flow. The user sends a request to execute a service. The request is received by the service director SD. SD initiates service execution by sending a message *send_req(request)* to SC. SC tries to execute the requested service. In case it completes its execution, it returns the message *success* to the service director. In case it fails, it sends a *failure* message to ER, i.e., requests to initiate error recovery. ER notifies SD about success *rec_result* or failure *unrec_failure* of error recovery.

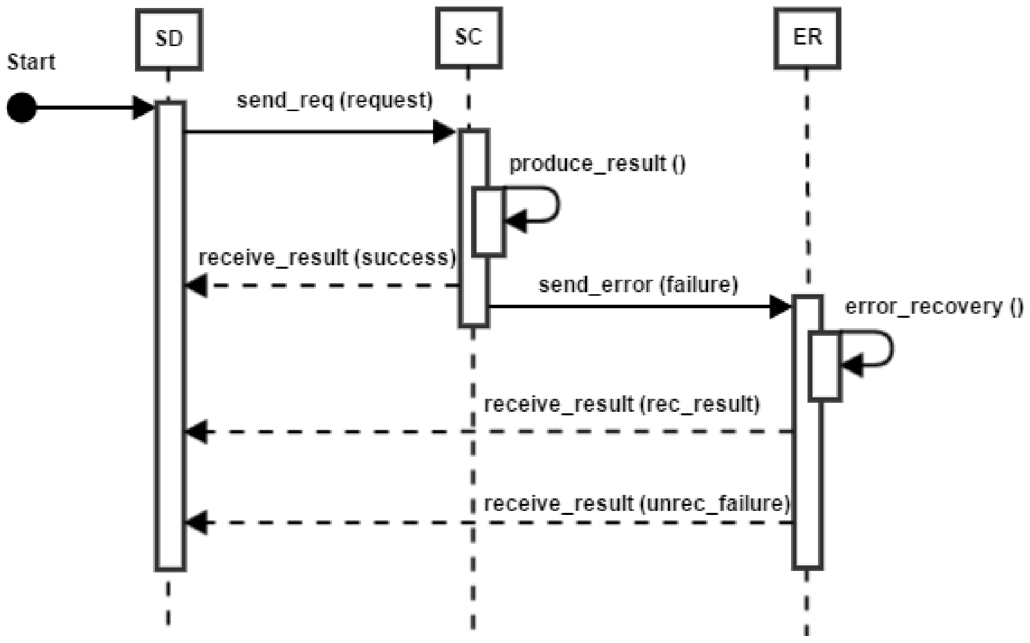
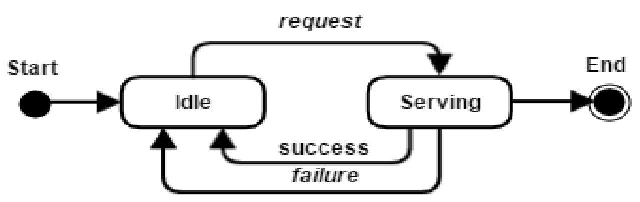


Figure 3.3: Example of a Sequence Diagram

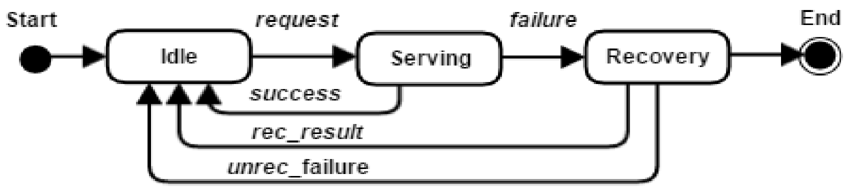
Another diagram representing the dynamic system behavior is the state diagram. The *state diagrams* represent the different states that a class or the entire system can be in and define the transitions from state to state. The state diagram has five basic elements: the initial starting point (a solid circle), a transition between states (a line with an open arrowhead), a state (a rectangle with rounded corners), a decision point (an open circle) and one or several termination points (a circle with a solid circle inside it).

An example of a state diagram defined on a system level is shown in Figure 3.4. The model depicts the communication between the service and its user. To request the service execution, the user generates an event *request* that is received by the service. The request is always replied either with *success* in the case of error-free system or *failure* in the case of error occurrence during the service provisioning.

In Figure 3.4 (a), the possibility of failed response is represented explicitly. This corresponds to the error detection step of fault tolerance implementation. The next extension would be to explicitly define the error recovery stage, as shown in Figure 3.4 (b). If the system is fault-free, then the service execution terminates normally and produces the required results together with the notification of success. If an error occurs, the system makes a transition to the recovery state. If the error recovery is successful, the system continues to operate normally; otherwise the *unrec_failure* event is generated designating that an unrecoverable error has occurred in the system.



(a)



(b)

Figure 3.4: Examples of State Diagram (a) with explicit representation of failure (b) with explicit representation of failure and error recovery

UML helps to analyze, visualize and discuss the design of software systems with different stakeholders. It is widely accepted in the industry due to its readability. However, as a graphical notation, it lacks formality. Nevertheless, it is a useful front end for a more rigorous notation, such as SystemC that we discuss next.

3.2 SystemC

SystemC [12, 13] is a widely used system-level modeling language [14] that allows the designers to represent the system design at different abstraction levels as well as use various design methodologies. The SystemC specifications can contain a mixture of abstract models and implementation-level code. The specifications are executable. Moreover, the framework supports simulation that allows exploring not only functional but also architectural alternatives.

SystemC provides us with a unified language to define both hardware and software components. SystemC-based design platforms help to cope with the complexity of system-level design and often result in reducing system development cost and time. SystemC has become de-facto verification standard in real-time embedded systems such as MARTE [17]. It is extensively used in industrial sectors such as electronic systems, semiconductor technologies, electronic design automation and embedded software.

SystemC is a high-level, powerful design language written in standard C++. SystemC uses C++ Class libraries and C/C++ Compiler (such as bcc, gcc, etc.). It introduces timing, concurrency and structure-related constructs to model system architecture. It provides capabilities similar to VHDL (Very-High-Speed Integrated Circuit Hardware Description Language) and strong simulation kernel that also supports hardware/software co-simulation.

The language of SystemC includes the following constructions: modules, ports, data types, processes, channels, interfaces, and events.

Modules are the basic building blocks of the language. They allow a designer to partition system design into smaller parts that encapsulate internal data representation and algorithms from the rest of the system. Hence, a SystemC model usually consists of several modules.

A typical module contains *ports* via which the modules communicate with each other, processes describing the functionality of a module, internal data, and channels representing model state and communication between processes.

A template for representing a module in SystemC is given in Figure 3.5.

```
SC_MODULE(Name) {
        // ports, processes, internal data, etc
         SC_CTOR (Name) {
                // body of construction:
                //process declaration,  event sensitivities etc.
                }
        };
```

Figure 3.5: Definition of simple SystemC module

The SC_MODULE macro designates that the class Name is derived from the library class sc_module. The macro SC_CTOR declares a constructor, which maps the membership functions to processes and declares event sensitivities. The argument should be the name of the module that is currently declared.

Modules can be instantiated, i.e., there might be several instances with the identical structure and functionality that are described by one common module definition.

SystemC uses interfaces, ports, and channels to provide the abstract modeling primitives for representing communication between modules. *Channels* are used to model the actual data transmission; *interfaces* describe the sets of operations that the channel provides. Finally, *ports* are the proxy objects that facilitate access to the channels through the interfaces. Often one can say that the port is bound to the channel through an interface.

An interface consists of a set of *operations*. For each operation, it specifies the operation's name, parameters and returns value. The abstract base class sc_interface is used to derive interfaces in SystemC. It also provides a virtual function register_port(), which is called to connect a channel with the port via the interface.

Let us consider an example of a simple interfaces sc_signal_in_if<S>. It is derived from sc_interface and parametrised by the datatype S. As such, it provides a virtual function read() that returns constant reference to S. Similarly, the interface sc_signal_inout_if<S> provides a virtual function write() that takes as parameter a constant reference to S. Since sc_signal_inout_if<S> is derived from sc_signal_in_if<S>, it also inherits the function read(). The semantics of these functions is to read from and write to the channel that implements the corresponding interface.

Interfaces significantly improve reusability of the modules that we create. The next step is supporting reusability to provide modules with the ports that allow them to connect and communicate with the rest of the system.

Ports correspond to interfaces and represent the objects derived from an abstract class sc_port_base. Figure 3.6 shows a template class for creating ports.

```
template<class IF, int N=1>
class sc_port : … // class derivation details
{
public:
        IF* operator ->();
        //member function, member variables
};
```

Figure 3.6: SystemC template for creating ports

The template class sc_port has two parameters: an interface IF through which the ports may be connected and an optional integer N denoting the maximum number of interfaces that may be attached to the defined port. The method of the port is operator ->(). It returns a pointer to the interface with which the port is associated.
For instance, we can map the following declaration of the port p into the definition of the module SC_MODULE given in Figure 3.5:

```
sc_port<sc_signal_inout_if<int> > p
```

The port accesses a channel [18] through the interface sc_signal_inout_if<int>. Since the interface provides the read() and write() methods, we can read the value of the channel and write to it using the expressions p->read() and p->write().

Process is a basic unit of functionality that is comprised of a module. Processes provide us with the means to simulate concurrent behavior, since in embedded fault-tolerant systems many activities are run in parallel.

A process is a member function of a module and is defined in the module construction. A defined process accesses external channels through the ports of its containing module. There are two kinds of processes: method process declared with the macros SC_METHOD and thread process declared correspondingly as SC_THREAD.

Figure 3.7 shows an example of a module with a method process declaration.

```
    SC_MODULE(Maximum) {
    sc_in<int>        n1;
    sc_in<int>        n2;
    sc_out<in>      max_n1_n2;

    void find_max() {
            if (n1>n2) then max_n1_n2  = n1
                            else max_n1_n2 = n2;
    }

    SC_CTOR (Maximum) {
    SC_METHOD (find_max);
    sensitive << n1 <<n2;
    }
};
```

Figure 3.7: Example of a module with a method process declaration

When the membership function find_max is invoked, it calculates the maximum of the inputs n1 and n2 and writes the result to the output max_n1_n2. However, the member function does not create a process as such. The process is created inside the constructor by the statement SC_METHOD(Maximum), which maps the member function to a method process by registering it with the scheduler. The next statement defines that this process is sensitive to changes in the values of signals that will be connected to the input ports, i.e., when n1 and n2 change, the value of max_n1_n2 is recalculated.

Next, we show how to describe the proposed architectural and behavioural the pattern for fault tolerance-explicit service design described in Chapter 3-1 in SystemC. The fragment of the code in Figure 3.8 illustrates that as a reaction to a service request received from a user, the service director sends a request to the service component. The service component can either succeed or fail to execute a service, i.e., it produces outputs *success* or *failure*.

The *class1* module can only call member functions of SD interface. Similarly, the *class2* module can only call member functions of SC interface. The process describes the functionality of the SD and SC in each respective module blocks. The SC receives a command from SD according to user's request (i.e., *request).* SC class generates output (i.e., *result* in the case of success, or *failure* in case of error). In the case of failure, error recovery is carried out, i.e., ER class produces *rec_result* in the case of recoverable error and *unrec_failure* otherwise. In the absence of faults, the service returns *success.*

```
class SD: public sc_interface { // SD class
public:
int request; };
class SC: public sc_interface { // SC class
public:
int result = 1; int failure = 0; int out_;
void output(const SD &sd){
if (sd.request == 1)
out_ = result;              // error-free
else
out_ = failure;             // fault
};
class ER: public sc_interface {    // ER class
public:
int failure = 0; int rec_result = 1;
int unrec_failure = 0; int out_;
void error(const SC &sc){
if (sc.out_ == failure)
out_ = rec_result;                 // recovered fault
else
out_ = unrec_failure; };           // unrecoverable fault
SC_MODULE(class1)   {              // module declaration
public:
sc_port<SD> req;                   // port
SC_CTOR(class1) {                  // constructor
SC_THREAD(main); }
void main(){                       // process
req.request = 1;}};
SC_MODULE(class2)   {              // module declaration
public:
sc_port<SC> out;                   // port
SC_CTOR(class2) {                  // constructor
SC_THREAD(main);}
void main(){                       // process
out.output();}};
```

```
SC_MODULE(class3) {            // module declaration
public:
sc_port<ER> out;               // port
SC_CTOR(class3) {              // constructor
SC_THREAD(main);}
void main(){                   // process
out.error();}      };
SC_MODULE(top) {               // top level serving module
public:
SC S_C; SD S_D; ER E_R;
class1* service_director;class2* service_component;
class3* error_recovery;
SC_CTOR(top) {                 // constructor
service_director = new class1("Service_Director");
service_director -> req(S_C);
service_component = new class2("Service_Component");
service_component -> out(S_D);
error_recovery = new class3("Error_Recovery");
error_recovery -> out(E_R);};
```

Figure 3.8: SystemC Implementation of Fault Tolerant Service Pattern

SystemC provides us with a systematic framework for modeling complex fault tolerant systems. However, as we have discussed previously, fault tolerance introduces additional complexity in the system design and hence increases the likelihood of design mistakes. To ensure correctness of fault tolerance mechanisms, we should employ the techniques used for fault avoidance, i.e., use formal verification. The most widely formal verification technique is model checking. Next, we describe it in details and present an example of properties that should be verified to guarantee the correctness of fault tolerant systems.

3.3 Model Checking Fault-Tolerant Systems

3.3.1 Model Checking: Introductory Background

Model checking is a verification technique for verifying functional properties of computing systems [98, 99, 100, 101,102]. Model checking takes as an input a finite-state model of the system and the desired property and systematically checks whether the given model satisfies this property. Typically, the properties that we are interested in checking include deadlock freedom, invariants, and request- response properties.

Model checking is an automated technique that allows us to check the absence of errors (i.e., property violations) in the system, i.e., to establish that the system under consideration has certain properties. In the majority of distributed systems including fault tolerant systems, we would at least be interested in establishing that a system never reaches a situation in which no progress can be made. In other words, more often than not, we are interested at least in verifying that no deadlock scenario can be found in the specification of the system.

Since the system specification defines the behavior of the system under consideration, the specification constitutes the basis for any verification activity. The system is correct only if it satisfies all properties derived from its specification. Once a state is found that does not satisfy one (or several) of the specification's properties, we can claim that we have discovered an error in the design. Therefore, correctness as such is relative to a specification.

Model checking is a verification technique that explores all possible system states in a brute-force manner. State-of-the-art model checkers can handle state spaces containing up to 10^9 states. The use of optimizing algorithms and abridged data structures sometimes allows the model checkers to tackle models containing up to 10^{20} states.

The system model is usually automatically generated from a model given in some programming language like C or modeling language like SystemC or Promela (which we present later). The model checker examines all system states to check whether they satisfy a particular property. If the model checker encounters a state that violates the property under consideration, it generates a counterexample indicating how the undesired state can be reached. Essentially, the counterexample shows an execution path leading from the initial system state to a state that violates the verified property.

A standard way to represent a model is by a transition system – a directed graph where nodes represent states and edges model transitions, i.e., state changes. A state represents some information about a system at a certain moment of its behavior. Formally, a transition system *TS* is defined as a tuple $(S,Act,\rightarrow,I,AP,L)$ where

- S is a set of states,
- Act is a set of actions,
- $\rightarrow \subseteq S \times Act \times S$ is a transition relation,
- $I \subseteq S$ is a set of initial states,
- AP is a set of atomic propositions, and
- $L{:}S \rightarrow 2^{AP}$ is a labeling function.

TS is called finite if *S, Act* and *AP* are all finite. In our definition, we describe the transition systems with the action names for the transitions (state changes) and the atomic propositions for the states. Intuitively, the behavior of a transition system is as follows: the system starts in some initial state $s_0 \in I$ and evolves according to the transition relation $\rightarrow$, i.e., if s is the current state, then a transition $s \rightarrow^{\alpha} s'$ originating from s is selected non-deterministically and taken. This means that the action α is performed and *TS* evolves from state s into the state s'. The procedure repeats until a state that does not have outgoing transitions is reached.

The labeling function L relates a set $L(s) \in 2^{AP}$ of atomic propositions to any state s, i.e., $L(s)$ identifies only those atomic propositions $a \in AP$, which are satisfied by state s. For a given propositional logic formula Φ, s satisfies the formula Φ if the evaluation induced by $L(s)$ makes the formula Φ true, i.e.,

$$s \models \Phi \text{ iff } L(s) \models \Phi$$

Let us consider a simple example – a road crossing. We have two traffic lights regulating the car movement in the orthogonal direction. Both traffic lights have two states: red and green. Both traffic lights are described by the similar transition systems consisting of two states: *red* and *green* and two transitions: *a* and *b* indicating the change of the light. For the first traffic light $red \rightarrow^{a} green$ and $green \rightarrow^{b} red$. For the second traffic light $red \rightarrow^{b} green$ and $green \rightarrow^{a} red$ correspondingly. The crossing can be represented as a parallel composition of two transition systems. Both traffic lights synchronize using the actions *a* and *b* indicating the change of the light. Let us assume that the initial state for both of them is red. Since the first traffic light is waiting to be synchronized on action *a,* the second light is blocked because it is waiting to be synchronized with action *b*. It represents deadlock and an obvious mistake in the system design.

Linear Temporal Logic (LTL) provides an intuitive but mathematically rigorous notation for expressing properties about state labels in the model execution. LTL formulae over the set AP of atomic propositions are formulated using the following grammar:

$$\varphi ::= true \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg \varphi \mid O \varphi \mid \varphi_1 \cup \varphi_2$$

where $a \in$ AP.
The until operator allows us to derive temporal modalities $\Diamond$ eventually, "sometimes in the future" and $\Box$ always, "from now on forever" as follows:

$$\Diamond\varphi = true \cup \varphi \text{ and } \Box \varphi = \neg \Diamond \neg \varphi$$

For instance, let us express the property that two processes *P1* and *P2* never simultaneously access their critical section (a shared resource):

$$\Box (\neg crit_1 \lor \neg crit_2).$$

This formula expresses that always $\Box$ at least one of the two processes is not in its critical section $(\neg crit_i)$. This is an example of a safety property. Another type of properties – liveness properties – describe the progress of the system. For our example of the critical section, the liveness property states that each process *Pi* is infinitely often in its critical section:

$$(\Box\Diamond\, crit_1) \land (\Box\Diamond crit_2)$$

There is a variety of general and specific model checkers that automate verification. Though they vary in their performance and underlying algorithms, they follow the same verification process as shown in Figure 3.9. The preprocessor extracts a state transition graph from a program, model checker takes the state-transition graph and a LTL formula and determines whether the formula is true or false.

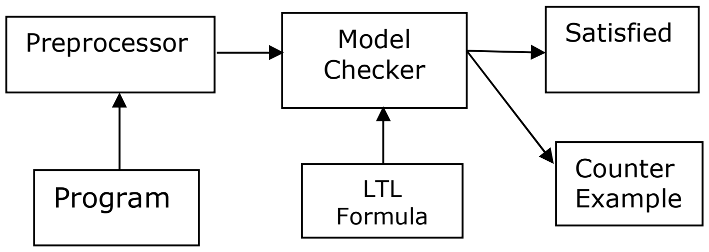


Figure 3.9: Model Checking Process

One of the popular model checkers is SPIN. Next, we give its overview and present examples of models of fault-tolerant systems.

3.3.2 SPIN PROMELA

SPIN [22, 66] is a model checker that has been initially developed to verify communications protocols but quickly became popular for verification of distributed and concurrent systems in general. The system models verified by SPIN are written in PROMELA – Process Meta Language. Correspondingly, SPIN stands for Simple PROMELA Interpreter (SPIN). It is a simulator and verifier for the properties of PROMELA models.

A program in PROMELA consists of a set of processes that can have parameters. SPIN compiles and executes PROMELA programs in the simulation mode. PROMELA adopts the syntax and semantics of C programming language to define expressions. The control statements are defined using the notion of guarded commands, which is particularly suitable for expressing nondeterminism inherent in distributed communication systems. There are five control statements: sequence, selection, repetition, jump, and unless.

The PROMELA programs can contain assertions – the statements containing predicates that are evaluated during the execution. When an **assert** statement is executed during a simulation, the predicate expression is evaluated. If it evaluates to true, execution proceeds normally, otherwise, the program terminates and gives an error message. Though assertions are useful for verification, it is limited to a verification of properties at specific control points in the processes.

To check the correctness of a model, we should verify that the desired properties hold through all possible execution paths. For the distributed and concurrent programs, it means that we have to verify all possible interleaving of the statements. When SPIN simulates a program, it creates one computation by interleaving the statements of all the processes. Typically, the design errors occur due to unforeseen interferences between processes. The use of LTL can significantly help in finding hidden interferences and debugging complex distributed and concurrent fault-tolerant systems.

SPIN supports verification of a standard set of temporal operators: eventually $\Diamond$ (denoted as <>), always $\Box$ (denoted as []), and until $\mathcal{U}$ (denoted as U).

Figure **3.10** presents the PROMELA model of the fault tolerant service architecture described in Chapter 3-1. SD process defines *request* command for SC and SC process generates the respective output depending on the *request*. If the output is a *success*, the system is error free. In the case of error, the error recovery is initiated. If error recovery succeeds, then *rec_result* is generated otherwise, *unrec_failure* is produced.

```
bool request, success, failure;
bool rec_result, unrec_failure;
bool ack = true;
active proctype SD() {
if
:: ack == true ->   request = true;
:: else
#ifdef FIX
-> break
#endif
fi;
}
active proctype SC() {
if
:: ack == true -> {
success = false; failure = true;
if
:: failure != false -> {
bool error_flag = true; run ER(error_flag);}
fi;}
:: else
#ifdef FIX
-> break
#endif
fi; }
proctype ER(bool e) {
e = false;
if
:: e == false -> rec_result = true;
:: e == true -> unrec_failure = true;
:: else
#ifdef FIX
-> break
#endif
fi;}
```

Figure 3.10: Promela Implementation of Fault-Tolerant Service

```
State-vector 24 byte, depth reached 14, errors: 0
37 states, stored
22 states, matched
59 transitions (= stored+matched)
0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
0.001  equivalent    memory       usage   for     states
(stored*(State-vector + overhead))
0.290  actual memory usage for states
64.000memory used for hash table (-w24)
0.343  memory used for DFS stack (-m10000)
64.539 total actual memory usage
```

Figure 3.11: Example of SPIN output

In our case, the verification was successful. In the case of deadlock, the output file shows a counterexample. The output file also depicts the number of states and transitions.

SPIN allows us to verify various properties expressed as LTL formulae. For instance, we can verify that eventually, service execution succeeds as follows:

$$\Diamond(\text{success} = \text{true})$$

Model checking is a popular verification technique. Its main advantages are as follows:

- It does not require sophisticated mathematical knowledge from the user. The user enters a description of the program to be verified and specifications to be checked. The checking process is an automatic "push-button" technology.
- Model checking is fast compared to other rigorous techniques such as proof-based verification.
- The model checker generates a counterexample in the case of property violations. A counterexample is a concrete execution trace that shows the error. The counterexamples are helpful in debugging complex systems.
- Model checking can be utilized during the design of a complex system with partial specifications. We do not have to wait until the design phase is complete.
- Temporal Logics can easily express many concurrency properties

However, as any brute-force technique, model checking suffers from the scalability problem known as state explosion.

In our work, we extensively relied on model checking to verify various aspects of fault tolerance. In particular, we used it to verify the functional correctness of mode-rich systems – a large class of systems that we overview next.

4 Fault Tolerance in Mode-Rich System

The notion of modes is widely used to structure the behavior of complex computer- based systems. Often analyzing system fault tolerance, we identify normal and degraded operational modes. This is an example of a very simple mode logic. For a large class of systems, mode logic is much more complex and is defined by a variety of internal and external conditions. In this chapter, we discuss the problems associated with designing fault-tolerant systems with complex mode transition schemes.

4.1 Modes

Modes are defined as mutually exclusive sets of system behaviors. Modes are extensively used to structure the dynamic behavior of systems [4] from different domains such as automotive, avionic and space.

The system modes are introduced based on the operational conditions of the system. Designing mode-rich systems is a complex engineering task. Firstly, the mode logic should be designed in such a way that the conditions triggering mode transitions are deterministic, and modes themselves are deterministically deducible from the states of the components. Secondly, for each mode, we should explicitly define mode entering conditions and mode invariant, which is preserved all the time while the system remains in this particular mode. Finally, mode logic should unambiguously define relationships between the modes at different layers of the system architecture.

Often structuring the system design according to the operational modes provides us with a suitable basis for introducing fault tolerance mechanisms. The main principle is to define the mode logic in such a way that every time upon the occurrence of a component failure the system makes a transition to a more degraded mode in which the failed component is inactive.

While designing the mode logic according to this principle, we introduce two types of transitions: forward and backward. The forward mode transitions bring the system from the current mode to the mode in which the system provides more advanced functionality. Correspondingly, backward mode transitions occur when an error occurs, and the system should be put into the more degraded mode compared to the current one.

Figure 4.1 illustrated the general principle described above. Since we present an excerpt from the overall mode logic, we assume that we start in the mode *current*. When there is no error in the current mode, then next mode is reached as a result of the forward transition. In the case of failure, in the current mode, the backward transition occurs that brings the system to a more degraded mode (previous mode) in which the failed component is not used, and hence the effect of its error is masked.

From the architectural point of view, we need to introduce the dedicated components that orchestrate mode transitions – we call such components *Mode Manager*s (MM). MM monitors the states of other system components that are controlled by unit managers (UM). When the conditions for entering another mode are satisfied, Mode Manager MM sends the commands to Unit Managers to make the transitions to the corresponding mode.

Typically, each particular Unit Manager UM controls several units (components). Often the unit consists of two branches – redundant and nominal, i.e., there are two components – the nominal and a spare. UM is responsible for managing the state transition of unit branches.

Let us assume that the initial system mode is Powering. It takes place when the unit branch state switches from off to on. Similarly, un-powering is carried out when the unit branch state switches from on to off. Failure Detection, Isolation and Recovery (FDIR) Manager handles the errors of the units and provides an error recovery algorithm to recover from faults. In the case of a nominal unit branch error, FDIR initiates unit reconfiguration by switching the failure branch (nominal) to backup branch (redundant). After unit reconfiguration, MM executes forward mode transition. When an error occurs in the redundant branch of the unit, the corresponding notification is sent to the FDIR manager, and the backward mode transition is trigged by MM.

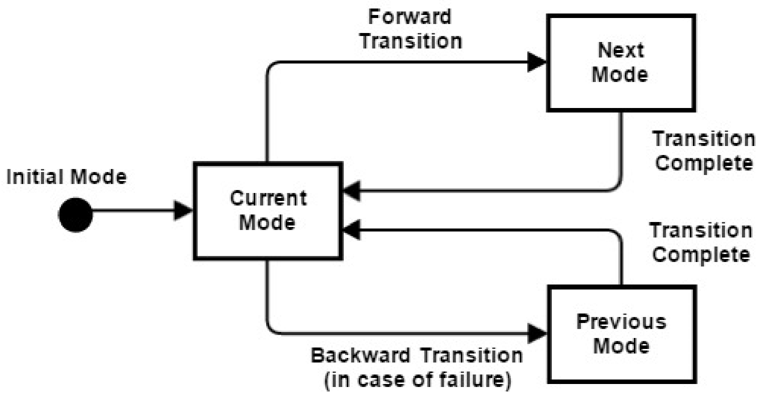


Figure 4.1: Mode Logic of Fault-Tolerant System: a General Principle

In the case of the centralized mode management, a single mode manager is responsible for controlling mode transitions. However, in the distributed mode management, several mode managers (i.e.; MM1, MM2, etc.) communicate with each other to implement the desired mode logic. The generic architecture of a fault tolerant with a distributed mode management is shown in Figure 4.1.2. Next, we consider two cases: a simpler one with one mode manager, i.e., the case of the centralized mode management and then a more generic one – with the distributed mode management.

4.2 Mode-Rich Fault-Tolerant Systems: Centralized Mode Management

We discuss the architecture and properties of a mode-rich fault-tolerant system with the centralized mode management by an example -- Attitude and Orbit Control System (AOCS) [5] [7]. The purpose of the Attitude and Orbit Control System (AOCS) is to acquire and maintain the attitude of a spacecraft throughout its mission. AOCS is activated upon the spacecraft separation from the launch vehicle. AOCS is an example of mode-rich systems with a complex mode transition scheme. AOCS is a generic component of different kinds of spacecrafts. Its development and verification have been undertaken by various teams, including the EU project DEPLOY [6]. AOCS performs multiple tasks such as sensor data processing, control computation and actuator commanding [8].

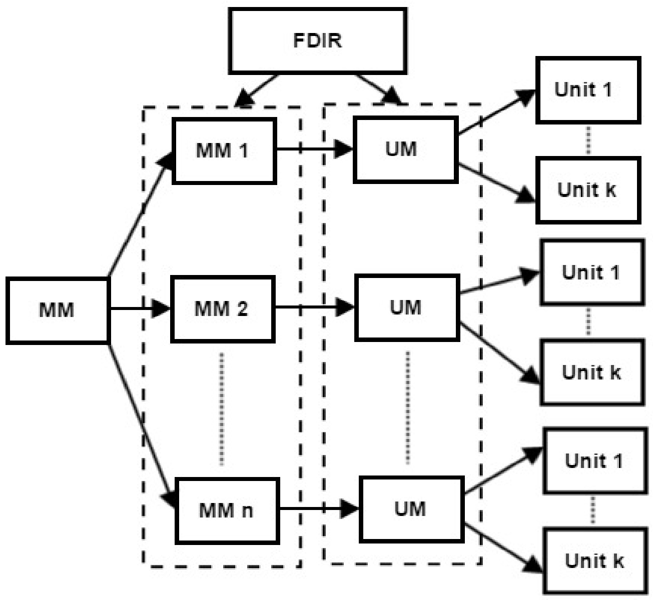


Figure 4.2: Generic Architecture of Mode Rich Systems

The main functionality of AOCS is to control the attitude and the orbit of a satellite. The satellite is continuously changing its positioning due to various environmental disturbances. However, to perform the scientific measurements, the satellite should maintain a certain accurate position for some period. Therefore, the attitude needs to be monitored and adjusted accordingly. AOCS consists of several components that manage the attitude and the orbit of a satellite as well as carry out scientific measurements specific to the particular mission of the satellite. AOCS instantiates the generic architecture given in Figure 4.2 as follows: there are four managers -- AOCS Manager, Unit Manager (UM), Mode Manager (MM) and FDIR Manager (FM).

Figure **4.3** represents the version of the architecture of AOCS with the centralized mode management.

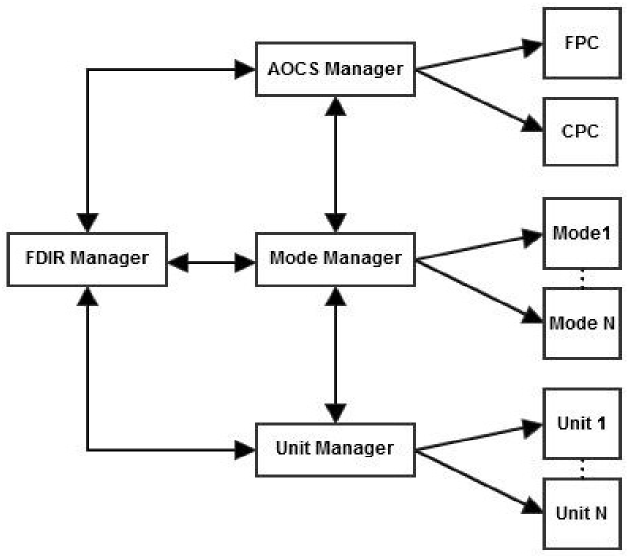


Figure 4.3: Centralized AOCS Architecture

The main operations of MM in AOCS are to check preconditions of mode transition, execute mode transitions, and manage the controllers and units. AOCS logic contains the following modes: Off (A), Standby (B), Safe (C), Nominal (D), Preparation (E), and Science (F) [8, 31]. The UM in AOCS is responsible for checking the states of the nominal branch (branch A) and redundant branch (branch B) of the units, to manage the state transition executions, and to handle the unit reconfigurations. Control Pointing Controller (CPC) and Fine Pointing Controller (FPC) are two control algorithms that are used to control computations. AOCS Manager manages them as well. Fault Manager – FM – handles errors of branch state transition and attitude errors. FM monitors error occurrence and initiates error recovery. An informal description of mode logic is given below:

1) All unit branches are put into the inactive state after a successful completion of mode transition to mode Off or mode Standby.
2) In mode Safe, the particular branches, i.e., Earth Sensor (ES), Sun Sensor (SS) and Reaction Wheel (RW) are set to ‘on’ state, while remaining branches go to ‘off’ state. Only Coarse Pointing Controller (CPC) is active in this mode.
3) In mode Nominal, GPS is set to ‘coarse’ state. While the particular branches, i.e., RW, Star Tracker (STR) and Thruster

(THR) are set to 'on' state and remaining branches go to 'off' state. Only Fine Pointing Controller (FPC) is active in this mode.

4) In mode Preparation, Global Positioning System (GPS) is set to 'fine' state and Payload Instrument (PLI) to 'standby' state, while the remaining unit branches and controllers maintain the same state as in the previous mode D.
5) After successful completion of mode Preparation, the satellite enters in mode Science. Only PLI is set to 'science' state in this mode, and remaining unit branches and controllers maintains the same state as in previous mode Preparation.

To visualize the AOCS mode logic, we use UML state diagrams. The detailed description of it is given in papers VI and VII included in part II of the thesis. Figure 4.4 shows the UML state diagram for GPS. The diagram depicts the transition from mode D to mode E and includes forward and backward transitions as well as unit reconfiguration.

GPS activation starts from mode D. If the GPS state is 'coarse' and there are no errors in other components, then forward mode transition occurs. If the GPS is in mode D, it remains 'off' for some period. If it does not change its state to 'coarse', then branch A and branch B of the GPS unit will be checked. When an error occurs only in branch A, then unit reconfiguration is carried out by deactivating the branch A and activating the branch B. After successful completion of unit reconfiguration, mode D is changed to mode E. The backward mode transition (change to mode C) takes place when both branches of the GPS have failed.

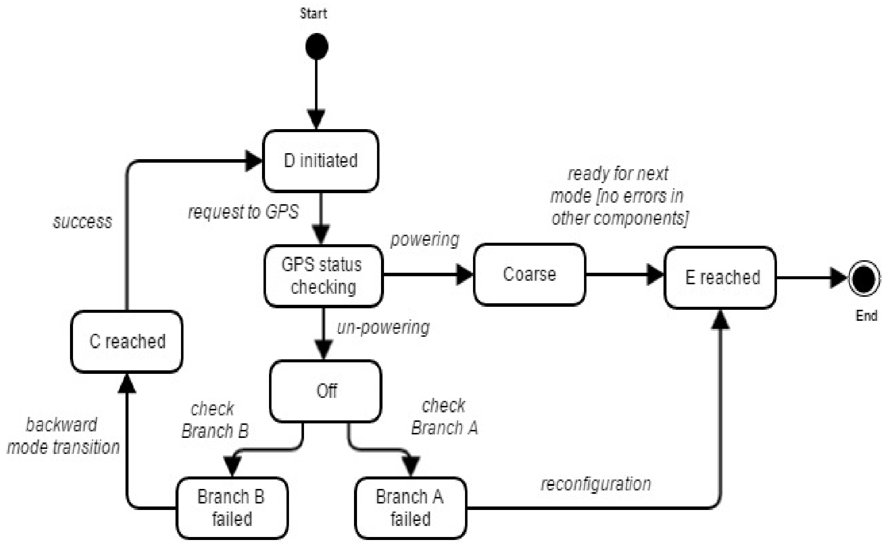


Figure 4.4: Centralized Mode-Rich Fault-Tolerant System: Example of GPS unit

The state diagram serves as a middle hand between the informal description of AOCS and its formal representation. Since AOCS is an example of a complex fault-tolerant embedded system, we used SystemC to implement and subsequently, verify it. The general pattern for modeling mode transitions in SystemC is given in Figure 4.5. For a given mode, it includes mode transition, error detection, and error recovery. We have chosen to illustrate it by the GPS example. The variable declarations include modes, statuses of GPS and its branches as well as the set of the UM commands to GPS. The status of GPS unit is checked according to the current mode. When the status of GPS satisfies the condition of the current mode, then MM switches the mode to the next mode. If it detects the branch state errors, then two cases are considered. In the first case, if the failure of branch A occurs but branch B is healthy, then the unit reconfiguration is performed. In the second case, if both branches have failed then backward mode transition is initiated.

```
Variables
prev_mode: {A, B, C, D, E, F}
curr_mode: {A, B, C, D, E, F}
next_mode: {A, B, C, D, E, F}
GPS_status: int
Branch_A_status: int
Branch_B_status: int

Begin

if GPS_status of curr_mode satisfied
then initiate a forward transition to next_mode
according to the predefined scenario;

if GPS_status of curr_mode failed
then check the nominal branch A and redundant
branch B of GPS;

if Branch_A_status of GPS failed
then initiate unit reconfiguration of curr_mode
and proceed to next_mode according to the
predefined scenario;

if Branch_B_status of GPS failed
then initiate a backward transition to prev_mode
according to the predefined scenario and the
choice of previous mode depends on the current
mode and its fault;

End
```

Figure 4.5: A general pattern for defining Mode Transition in SystemC (GPS instance)

To verify the correctness of our SystemC implementation, we used SPIN. Figure 4.6 illustrated the verification procedure. Upon powering UM, MM sends its current mode acknowledgment to UM, then UM sends a command to GPS to perform the necessary actions. As soon as they are completed, UM sends an acknowledgment. If the current mode is mode C and mode transition to mode D is going to take place, then UM authorizes it to send the powering request to GPS. In case GPS powering is true, then both branches (i.e., A and B) change their state to 'coarse' according to the mode D and proceed to the next mode E. In the case of a fault, error recovery is initiated.

SPIN checks the absence of deadlocks and livelocks. As it is easy to see, to implement the desired mode logic, AOCS relies on a complex communication scheme. The use of model checking allows us to verify the correctness of its implementation and significantly improves our confidence in the validity of the developed system.

```
// initial status of GPS unit N_GPS = off;
R_GPS = off;
Powering = true; // powering for Mode D
if
::(N_GPS == coarse && R_GPS == coarse)→{
error_flag = 0;
run go_to_modeE(N_GPS, R_GPS); }
::(N_GPS != coarse && R_GPS == coarse)→{
error_flag = 1;
run reconfiguration(N_GPS, R_GPS);
run go_to_modeE(N_GPS, R_GPS); }
::(N_GPS != coarse && R_GPS != coarse)→{
error_flag = 1;
run go_to_modeC(N_GPS, R_GPS); }
:: else
#ifdef FIX
→break
#endif fi;
```

Figure 4.6: Example of Mode Transition Verification

We have considered the case of a centralized mode management. In this case, the architecture contains one dedicated component – Mode Manager – that has a "global" view on the system state and hence is capable of making the decisions about mode transitions.

Often satellite systems are composed of several rather independent subsystems. Each of the subsystems has its mode manager. Therefore, to make the decisions about “global” mode transitions, the mode managers, which are distributed across the system, should communicate with each other and agree on a common decision regarding the next mode.

Next, we consider this case, i.e., discuss fault tolerance of mode-rich systems with the distributed mode management.

4.3 Mode-Rich Fault-Tolerant Systems: Distributed Mode Management

Distributed mode management introduces additional complexity into the design of fault-tolerant systems. Let us consider the following scenario. We have two subsystems, and each of them has its mode manager. Assume that each subsystem has reached the conditions to switch to a more advanced operational mode. However, while making such a transition, an error occurs in one of the systems, and as a result, its local mode manager issues a command to switch to a more degraded mode. Obviously, the system would be left in an inconsistent state if before issuing a command to switch to a new mode the local mode managers would not agree on the common mode transition.

Such an agreement is often called a handshake protocol. The handshake is a process in which connection is established between two modules, and the information is transferred from one module to another. As a result, two modules agree on a common action (or data).

Let us discuss how to use the handshake protocol to ensure the correctness of mode logic of fault tolerant systems with the distributed mode management. We consider a system that consists of two identical modules [32]. When one of the components of the first module fails, the system switches to the use of the second module. However, if at a certain point in time, some component of the spare module fails too, then none of the models remains operational. To avoid failure of the entire mission, the system should employ healthy components of both modules and define the desired control flow over them by introducing a communication between the modules and their corresponding mode managers.

Let us consider a modified version of AOCS – Distributed Attitude and Orbit Control System (D-AOCS). In D-AOCS, mode transitions and error recovery involve a sophisticated synchronization between mode managers – the synchronization that is implemented using a complex handshake protocol.

D-AOCS contains two mode managers – MM1 and MM2, which manage mode transitions in their respective parts. The mode managers execute mode transitions in parallel and use handshaking to synchronize on the phases of changing between modes.

The mode managers proceed according to the following generic behavioral pattern. Assume that the conditions for entering next mode are satisfied in module 1. Mode manager MM1 sends a request to MM2 with a proposal to enter the next mode. The Mode manager of the second module, MM2, checks the state of the module. If the conditions for entering the proposed mode are satisfied, then it sends a reply-designating that it has agreed on the transition to the proposed mode. Upon receiving the reply, the first mode manager issues the command to the components of module 1 to make the transition to the desired mode. Moreover, it sends the acknowledgment to MM2 that the transition to the next mode has been initiated. Upon receiving such an acknowledgment, MM2 issues the command to the components of Module 2 to make the corresponding mode transition. Then it sends the acknowledgment to MM1. Therefore, both modules agree (i.e., "handshake") on the mode transition. As a result, the components of both modules are managed in a synchronized way.

Both forward and backward mode transitions follow the behavioral pattern described above. The only difference is that the mode manager of the module that has experienced an error initiates the backward mode transition. Informally, the requirements of D-AOCS can be described as follows:

1) When both modules enter mode A or B, all units and controllers have inactive/idle status. MM1 and MM2 periodically communicate with each other to notify that there is no error occurred, and the given modes are maintained. After successful acknowledgment, both managers switch their current mode to the next mode.
2) When mode transition to mode C is commanded by the managers, all units except ES, SS, and RW remain in 'off' state, and only CPC is activated. After that, both mode managers confirm error-free state to each other. MM1 and MM2 go to next mode after a successful handshake.
3) Upon successful transition to mode C, the modules should try to achieve mode D. The units – RW, STR, and THR are set to 'on' state, GPS is put in 'coarse' state. FPC is activated. When the desired states are reached in both modules, the mode managers handshake and become ready to progress to the next mode.
4) To enter mode E, only GPS and PLI should change their status to 'fine' and 'standby' respectively and the remaining units and controllers maintain the same status as in mode D. Then managers check that conditions for entering mode E are satisfied. Upon confirming that the conditions have been indeed achieved, the mode managers handshake and make a transition to the next mode.

5) In the case of the mode F, the PLI unit operates with ‘science’ state. All other units and controllers keep the states reached in mode E. MM1 and MM2 both update each other regarding the success of mode transition and stay in this mode until all desired operations of the satellite’s mission are accomplished or an error occurs, and a backward transition is initiated.

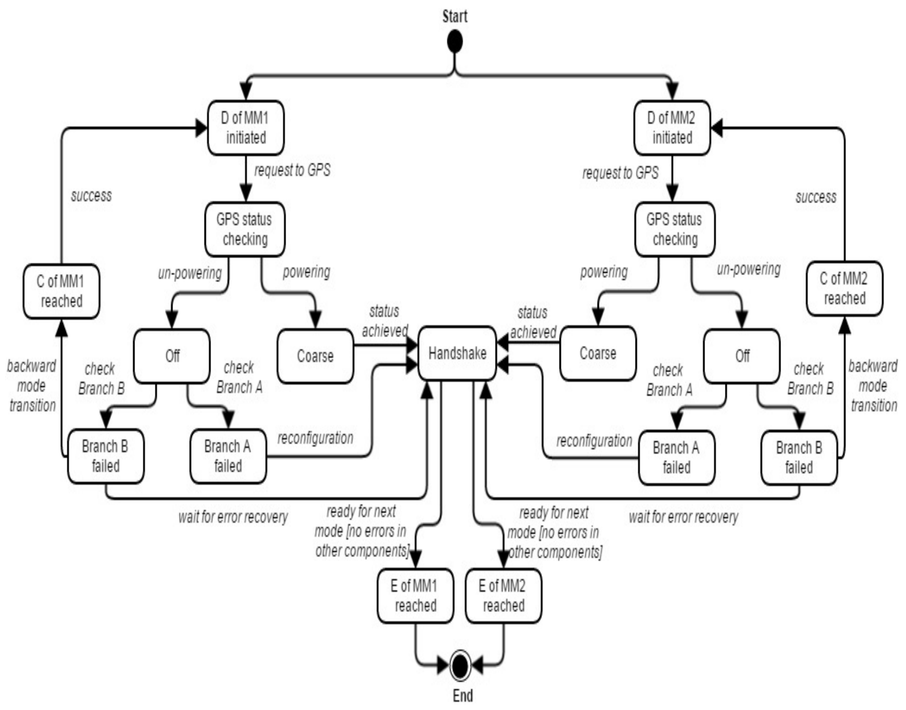


Figure 4.7: State Diagram of D-AOCS (GPS example)

The state diagram shown in Figure 4.7 defines the dynamic behavior of the GPS unit of D-AOCS. The diagram shows the mode transition from mode D to E and from mode D back to C. GPS activation starts from mode D by both managers. If GPS status in mode D of MM1 or/and MM2 remain inactive, then branches of GPS unit will be checked to detect the faulty branch. Mode transition cannot be made until error recovery (reconfiguration) is completed. Upon that the mode managers handshake and proceed to the next mode.

If an error occurs only in branch A, then unit reconfiguration is carried out by switching off branch A and activating branch B. Backward mode transition takes place in MM1 or/and MM2 to mode C when branch B of GPS has failed. After successful completion of unit reconfiguration or backward mode transition, the mode managers send the response notification, and the handshake protocol is initiated. As a result, the mode transition from mode D to mode E can be initiated by both mode managers. The detailed modeling and verification of the handshake protocol in D-AOCS are described in Paper III.

A fragment of the general scheme implementing distributed mode management using the handshaking protocol is given in Figure 4.8. If GPS state in each module fulfills the requirements for entering the next mode, then mode managers notify through handshake procedure that the modules are in the error-free state and transition can be initiated. It results in the forward mode transition. If GPS error occurs in any of the modules, then the mode manager responsible for controlling the failed unit initiates an error recovery, i.e., the backward mode transition or the unit reconfiguration. The other mode manager waits until the error recovery of the failed unit is completed. After the successful error recovery, both mode managers proceed to the next mode simultaneously using the described handshake protocol. The detailed modeling and implementation of handshaking protocol in case of Control and Data Management Unit (CDMU) are explained in Paper IV.

The implementation of the handshake procedure for D-AOCS is verified using SPIN. We have checked the correctness of backward mode transition scenarios, forward mode transition scenarios, unit reconfiguration scenarios and handshake procedure.

Figure 4.9 presents an excerpt of the model describing the behavior in mode D. If GPS maintains the required state in both modules, then mode managers notify each other through the handshake procedure and initiate forward mode transition to the next mode E. If an error occurs during a mode transition in any module, then the mode manager responsible for controlling failed unit initiates an error recovery, i.e., the backward mode transition or the unit reconfiguration. Verification demonstrates that the system can deadlock if error recovery does not terminate. To ensure that the deadlock is excluded, we can introduce a deadline to guarantee that the error recovery cannot proceed indefinitely.

In this section, we described our approach to modeling and verification of mode-rich fault-tolerant systems. Such an approach can be used in developing distributed embedded systems where we can define a finite set of states for each component. Another large class of distributed systems is service-oriented systems. The use of mode-based modeling would be cumbersome for such kinds of systems, since the behavior of services is described via their inputs and outputs rather than states. Hence, to develop service-oriented fault-tolerant distributed system, we need to rely on other techniques. We describe them in the next chapter.

```
Variables
prev_mode: {A, B, C, D, E, F}
curr_mode: {A, B, C, D, E, F}
next_mode: {A, B, C, D, E, F}
MM1_GPS_status: int
MM2_GPS_status: int
MM1_Branch_A_status: int
MM2_Branch_A_status: int
MM1_Branch_B_status: int
MM2_Branch_B_status: int

Begin

if MM1_GPS_status and MM2_GPS_status of curr_mode satisfied
then initiate a forward transition in MM1 and MM2 to next_mode according to the
predefined scenario after handshaking;

if MM1_GPS_status or MM2_GPS_status of curr_mode failed
then check the nominal branch A and redundant branch B of GPS in current modes of
MM1 and MM2;

if MM1_Branch_A_status of GPS failed
then MM2 maintains the current mode and MM1 initiates unit
reconfiguration according to the predefined scenario and after reconfiguration
both managers initiate a forward transition after handshaking;

if MM2_Branch_A_status of GPS failed
then MM1 maintains the current mode and MM2 initiates unit
reconfiguration according to the predefined scenario and after reconfiguration
both managers initiate a forward transition after handshaking;

if MM1_Branch_B_status of GPS failed
then MM2 maintains the current mode and MM1 initiates backward transition to
prev_mode according to the predefined scenario. The choice of previous mode
depends on the current mode and its fault. After error recovery in MM1, both
mode managers initiate a forward transition after handshaking;

if MM2_Branch_B_status of GPS failed
then MM1 maintains the current mode and MM2 initiates backward transition to
prev_mode according to the predefined scenario. The choice of previous mode
depends on the current mode and its fault. After error recovery in MM2, both
mode managers initiate a forward transition after handshaking;
End
```

Figure 4.8: Distributed Mode Management (GPS example)

```
Powering = true; //powering for Mode D in MM1 & MM2
// GPS1 denotes for MM1 and GPS2 denotes for MM2
if
::(GPS1 == coarse && GPS2 == coarse)→{
MM1_error_flag = 0;
MM2_error_flag = 0;
run handshake(MM1_error_flag, MM2_error_flag);
run go_to_modeE(GPS1, GPS2); }
::(GPS1 != coarse && GPS2 == coarse)→{
MM1_error_flag=1;
MM2_error_flag=0;
run handshake(MM1_error_flag, MM2_error_flag);
if
::(N_GPS1 != coarse && R_GPS1 == coarse)→{
run reconfiguration(N_GPS1, R_GPS1);}
::(N_GPS1 != coarse && R_GPS1 != coarse)→{
run go_to_modeC(N_GPS1, R_GPS1);
run go_to_modeD(N_GPS1, R_GPS1);
MM1_error_flag = 0;}
fi;
run handshake(MM1_error_flag, MM2_error_flag);
run go_to_modeE(GPS1, GPS2); }
::(GPS1 == coarse && GPS2 != coarse)→{
MM1_error_flag = 0;
MM2_error_flag = 1;
run handshake(MM1_error_flag, MM2_error_flag);
if
::(N_GPS2 != coarse && R_GPS2 == coarse)→{
run reconfiguration(N_GPS2, R_GPS2);}
::(N_GPS2 != coarse && R_GPS2 != coarse)→{
run go_to_modeC(N_GPS2, R_GPS2);
run go_to_modeD(N_GPS2, R_GPS2);
MM2_error_flag = 0;}
fi;
runhandshake(MM1_error_flag,MM2_error_flag);
run go_to_modeE(GPS1, GPS2); }
fi;
```

Figure 4.9: Distributed Mode Management: Verification of Handshake Procedure (Mode D example)

5 Fault Tolerance in Service-Oriented Distributed Systems

In this chapter, we discuss the problem of ensuring fault tolerance of service-oriented systems. This is a large class of distributed systems, which includes a variety of networked applications. Ensuring reliable operation of such systems relies on dedicated architectural solutions as well as systematic analysis of failure modes of system components. In this section, we discuss our approach to developing fault tolerant service-oriented systems.

5.1 Fault Tolerance in Service Oriented Architectures

While ensuring fault tolerance, modern software should cope with an inherent heterogeneity of networked systems that often work over diverse platforms and rely on a variety of protocols as well as interact with a wide range of devices. Service Oriented Architecture (SOA) enables seamless integration of heterogeneous components and provide the developers with a powerful mechanism to cope with the complexity of such heterogeneous environment.

SOA is an architectural style that relies on loosely coupled interacting software components providing services. A service can be defined as a piece of functionality that is offered by a service provider to a service consumer. All service-oriented systems follow the same behavioral pattern: a service consumer sends a service request to a service provider, the service provider, either by itself or in cooperation with other service providers, executes the requested service to produce the required result and responds to the service consumer.

SOA relies on two main architectural principles to support loose coupling of the components. Firstly, interfaces should be defined for all interacting services. The interfaces should be available for all service providers and service consumers. Secondly, all messages are described via an extensible scheme; all messages are delivered through the interfaces.

The service interface can be seen as a contract defining the functionality of the service in a generic platform-independent manner. Therefore, the invocation mechanism including the protocols, service descriptions, and service discovery facilities should follow the predefined common standards.

Services should be self-describing, i.e., they should publish their capabilities, interfaces, behavior, and quality attributes. The description of the interface should include a service signature that includes its input, output, and error messages. The quality of service (QoS) attributed typically describes non-functional attributes, such as average response time, failure rate, etc. Moreover, it often includes a description of security policy.

While developing the methods allowing us to guarantee fault tolerance of service-oriented systems, we should take into account the following constraints. Firstly, the services are stateless, i.e., we cannot assume that the current conditions or internal states of a service are known. Secondly, since the services are loosely coupled, they do not share common modules. Finally, services should be location-transparent, i.e., we cannot assume that a certain service would be available locally.

SOA supports the construction of the complex services by coordinated aggregation of lower level services (subservices). The service director communicates with subservices. It orchestrates the execution flow of the service and coordinates error recovery, as explained in Chapter 3.

The important goal of introducing fault tolerance in the service architecture is to prevent a propagation of faults to the service interface level (i.e., to avoid a service failure [90, 91]). While we cannot observe the states of the subservices, we can monitor their responses and, in the case of subservice failure, rely on redundancy to mitigate the effect of error occurrence.

As we described in Chapter 2, to achieve fault tolerance, we always have to introduce some form of redundancy into the system architecture. We propose three types of architectural patterns for introducing redundancy into the architecture of complex aggregated services, as shown in Figure 5.1.

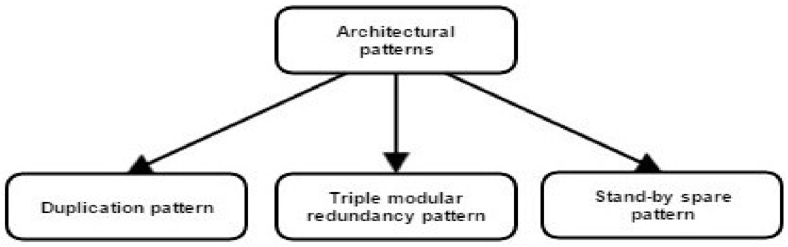


Figure 5.1: Architectural Patterns for Introducing Redundancy

The duplication pattern relies on introducing a redundant service component that provides the (sub)service that is identical to a certain subservice included into the service architecture. Both services can be executed in parallel. A successful execution of a service by any of two service components is sufficient to guarantee the reliable execution of the aggregated service.

The triple modular redundancy pattern implements the principle described in Chapter 2-2. The precondition for applying this pattern is that there are at least three service components available over the network that provide an identical (sub)service. To use the triple modular redundancy pattern, we also have to introduce a specific voting service into the architecture of the aggregated service. To execute a triplicated (sub)service, the request is sent to all three service providers. The results of the service execution are sent to a voting element – a dedicated software component that performs a comparison of the results and produces the final result.

The Stand-by spare pattern is an example of dynamic redundancy. It assumes an availability of a spare service component providing the desirable service. The spare is used only if the execution of the service by the main component fails. If the main service component succeeds in executing a service, the spare service component remains inactive.

The dynamic behavior of an aggregated service with the integrated fault tolerance mechanisms is shown in Figure 5.2. Here S1, S2,....Sn designates services from which the service is composed. IN and OUT correspond to a service request and service result. The execution flow might include repeated execution of a certain subset of subservices as well aborting the whole service.

If a transient error occurs in the service execution, then the service director can recover from the error by re-executing the failed subservice. If the error is unrecoverable and there are no possible redundant service providers available, then the service director aborts the entire service execution, and a failure response is returned. Service execution in case of individual service Si (where *i* is 1 to n) might include parallelism, though overall high-level service execution flow is considered sequential [36].

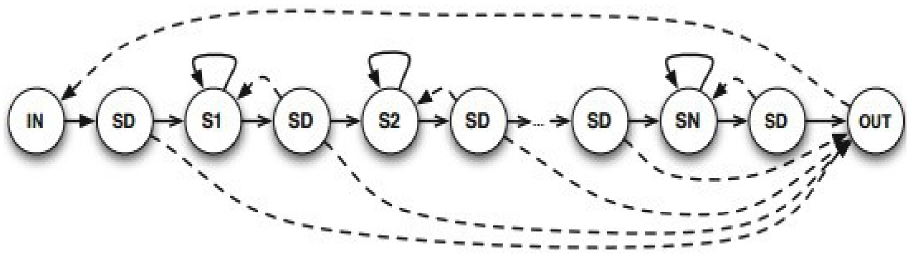


Figure 5.2: Service Execution Flow

To ensure that fault tolerance mechanisms are introduced into the service architecture in an appropriate way, we should support a systematic analysis of possible failure modes of the service. We propose to use Failure Modes and Effect Analysis (FMEA) technique to achieve this goal.

FMEA [76, 77, 78, 84] is an inductive analysis method that provides information about possible failure modes of the system and its components. It supports a systematic description of the error detection and recovery procedures. FMEA is used to investigate the causes of faults, their effects on the entire architecture and the means to cope with them. The results of FMEA are usually presented in the tabular form. Typically, an FMEA table contains the following fields: component name, failure mode, possible cause, local effect, system effect, detection, and remedial action.

FMEA can be performed at different levels, e.g., system level (focusing on global functions) or component level (inspecting functions of the individual component). The content of an FMEA table may vary. For example, it may take into account the probability of failure occurrence, severity, risk level, etc.

Figure 5.3 depicts the main fields of an FMEA table and their description.

Component	Component name
Failure mode	Potential failure modes
Possible cause	Reason associated with supposed failure mode
Local effects	Changes in the performance of a component
System effects	Changes in the performance of a system
Detection	Methods for failure mode detection
Remedial action	Actions to tolerate the failure

Figure 5.3: Generic Form of FMEA Table

Let us exemplify our approach to applying FMEA to systematically analyze failure modes in the context of SOA. Let *S1* be a subservice in the composite service *S*. Moreover, we assume that the failure modes of *S1* can be either a transient silent failure (no response) or detectable failure (failure response). In the case of the transient silent failure, the failures can be detected by timeout. The other failures are detected by receiving the failed response from the service.

To mitigate the failure of *S1*, we can use the triple modular redundancy pattern. However, we should check that there are two other service components available that provide services identical to *S1*. Figure 5.4 illustrates an analysis of the transient silent failure.

Component	S1
Failure mode	Transient silent failure
Detection	Timeout
The use of redundancy	Available
Available redundancy	S11, S12
Redundancy pattern	Triple modular redundancy arrangement.
Effect of redundancy	Result is produced upon timeout
Recovery	Masking failure by use of triple modular redundancy arrangement. In case of simultaneous failure of S1, S11 and S12 repeat execution

Figure 5.4: Example of Failure Analysis using FMEA

In our approach to architecting fault tolerant services, we have focused on analyzing input/output behavior of the services and demonstrated how to augment service architecture with various redundancy arrangements. In such an approach, the decision about embedding fault tolerance is done at the design time. However, often, during system development, our knowledge about system behavior is incomplete. By integrating capabilities to monitor system behavior at run-time and introducing the possibility to dynamically adapt system architecture, we can achieve a more flexible and efficient implementation of fault tolerance. Next, we discuss an architectural pattern that we proposed to develop adaptive fault tolerant systems.

5.2 Adaptive Fault-Tolerant Systems

Modern distributed systems should have a high degree of reliability, while efficiently utilizing the available resources. This is achieved by endowing the systems with the capabilities to dynamically adapt to changing external and internal conditions at run time. To ensure that the system architecture has a high degree of plasticity and can dynamically re-configure, we have to integrate into the system design the capabilities to monitor its behavior and environment at run-time, recognize symptoms of upcoming failure and proactively reconfigure to preclude failure occurrence of mitigating the failure impact.

The idea of proactive fault tolerance [82] comes from different domains such as autonomous computing, adaptive memory management and classic research on reliability engineering. Proactive fault tolerance encompasses three main steps: failure prediction, proactive reconfiguration, and recovery.
Failure prediction aims at spotting precarious situations, i.e., the states of the systems that will probably evolve into a failure. Failure prediction produces an estimate of how likely the current situation will result in a failure.

Proactive reconfiguration relies on the outcome of failure prediction. Based on the results of failure prediction, the system should make a decision about implementing the countermeasures that should be executed to rectify the problem. Usually, such decisions utilize some objective function that takes into account the cost of the actions (in terms of time, computational resources, memory and so on), the confidence in the prediction, and the effectiveness and complexity of the actions. As a result, it allows the system to determine the optimal trade-off. The generic problems associated with implementing the planned actions include the online reconfiguration of globally distributed systems, data synchronization of distributed data centers, and much more.

Recovery enables graceful degradation of services in case the resources are insufficient for precluding the failures. For example, if the proactive reconfiguration progresses more slowly than expected, then the system should compensate for the insufficient resources. Another example would be unexpected multiple failures of several components due to sudden adverse change in the environment.

To enable a systematic development of adaptive fault tolerant systems, we propose the pattern depicted in Figure 5.5 that ensures architectural plasticity and supports dynamic reconfiguration. The patterns allow us to define the system structure in a layered manner [83]. Each layer is responsible for a certain aspect of the system behavior.

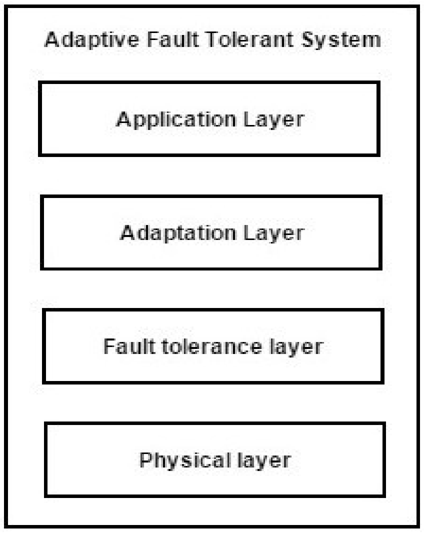


Figure 5.5: Architecture of an Adaptive Fault-Tolerant System

The physical layer represents the environment whose state should be monitored. It might be a complex control system that uses logical sensors to monitor the health of its components. Another example might be a sensor network that monitors the desired parameters of the system environment using physical sensors.

The fault tolerance layer performs the data aggregation and evaluation of the quality of monitoring. This information is supplied to the adaptation layer that is responsible for defining the proactive adaptation policy.

The aim of the application and fault tolerance layer is to continuously supply the application with the monitoring data of an acceptable quality. The design of the application is defined by its purpose – it varies from the complex control functions to collecting data intelligence.

Each (logical or physical) sensor produces data about the monitored parameter that includes the measurement and timestamp, i.e., the reference to the time when the measurement is taken. By supplying each measurement with the time stamp, we can detect "stuck at value" type of the sensor failure. By introducing a feasibility check, i.e., checking that the value falls within a certain physically reasonable range, we can detect the "incorrect value" type of the sensor failures.

The fault tolerance manager periodically reads the sensor data, filters out faulty data and computes the average value over the valid data set. Moreover, it defines the quality level of the produced data. Essentially, it checks that the computations of the estimate of a certain parameter are performed over the sufficiently large set of data. There are certain thresholds, i.e., the data sizes that allow the fault tolerance manager to estimate the quality level.

The adaptation manager and deployment manager constitute the adaptation layer. Their task is to determine whether the decline in quality of data are temporal or permanent. In the first case, no reconfiguration actions should be performed. In the second case, the reconfiguration actions should be initiated. Moreover, their scale is determined by the quality level of the produced data set.

The dynamic behavior of the system can be structured using the notions of modes introduced in Chapter 4. The generic representation of the mode transition logic is given in Figure 5.6.

```
Procedure ModeTransition
Variables
last_mode: {Normal, Adapt, Adapt_Compl, Adapt_activ}
next_target: {Normal, Adapt, Adapt_Compl, Adapt_activ}
prev_target: {Normal, Adapt, Adapt_Compl, Adapt_activ}
level: int

Begin
if adaptation completed
then initiate a forward transition to next_target according to
the predefined scenario;

if level dropped
then initiate a backward transition to next_target adaptation mode
The choice of target mode depends on severity of level decrease;

if the conditions for entering the target mode are satisfied
then complete a transition to next_target mode and become stable ;

if neither the conditions for entering
the next global mode are satisfied nor the level dropped
then maintain the current mode
End
```

Figure 5.6: Transitions between Adaptation Modes

The main principle that underlies the mode transition is as follows: the mode is stable and unchanged until a fluctuation in the quality level is registered. The algorithm describes *next_target* and *prev_target* modes in such a way that if adaptation procedure is completed, then the current mode switches to *next_target*. Similarly, if the quality level has dropped, then the current mode switches to *prev_target*, i.e., performs a backward transition. In the case of the quality level deterioration, the adaptation manager starts the remedial actions. When the system achieves an appropriate level of quality, then *last_mode* is reached, and the system becomes stable.

The proposed approach allows the designers to derive a clean architectural structure and achieve a separation of concerns. The information flow in the proposed layered architecture enables adaptation and guarantees a continuous delivery of service with an acceptable quality level. To preclude disruption in the provision of the services, an adaptive fault tolerant system structured according to the proposed architectural pattern performs a preventive reconfiguration.

In Chapters 2-5, we have presented a general overview of the methods and techniques that we used for the development and verification of fault-tolerant systems and described our contribution to creating a fault-tolerance explicit development approach. Next, we give an overview of the original publications in which the presented research has been reported.

6 Summary of the Original Publications

The chapter presents a brief summary of the publications included in the second part of this thesis.

Paper I: Implementation of SPIN Model Checker for Formal Verification of Distance Vector Routing Protocol

This paper presents the approach to modeling and verification of Distance Vector Routing (DVR) Protocol in the SPIN model checker. We demonstrate how to model the protocol and verify its correctness. The paper discusses the general methodology for implementing the distributed protocols relying on distance routing calculation. The DVR protocol is specified in PROMELA. The verification of correctness of DVR relies on the use of the SPIN model checker. The evaluation results suggest that the performance of the implemented protocol can be enhanced by reducing storage space requirements. Moreover, the reliability of the protocol can be improved by utilizing timing redundancy in case of failure of calculations, i.e., repeating the calculations in case of a transient error. Moreover, to improve protocol security, integration of mechanisms for message encryption and authentication can be implemented.

The main goal of this paper has been to study the principles of modeling and to verify complex distributed systems in SPIN. This paper has been specifically focusing on modeling the communication aspect of distributed systems.

Paper II: Designing a Fault-Tolerant Satellite System in SystemC

The paper presents an approach to designing fault-tolerant satellite systems in SystemC. Our goal was to study the principles of implementing mode-rich systems in SystemC. The paper gives a detailed description and implementation of an Attitude and Orbit Control System (AOCS). AOCS controls the attitude and orbit of a satellite. The system consists of a number of components with different functionality. Each component is represented by a corresponding module in SystemC. A dedicated component – Mode Manager which is responsible for monitoring the states of system components and coordinating mode transitions.

AOCS should implement a certain scenario defined in terms of mode transitions. The main goal is to reach and stabilize in the most advanced mode. In such a mode, the satellite can perform the scientific experiments and fulfill the goal of the mission. However, errors might trigger transitions to the more degraded mode, i.e., cause backward mode transitions.

In our SystemC implementation, we demonstrate how to coordinate both forward and backward mode transitions as well as perform system reconfiguration. We propose the patterns for defining components of mode-rich systems, specifying mode managers and unit managers, which are responsible for unit reconfiguration. We show how to define the overall architecture of a distributed system and the communication between the components. Our SystemC implementation is used not only to describe the design of the system but also as an input for verification.

Paper III: Modelling a Fault-Tolerant Distributed Satellite System

In this paper, we continue a study of mode-rich fault tolerant systems. In paper II, we considered the case in which control over the mode changes is performed by a centralized mode managing component.

In paper III, we consider the case of a distributed mode management. Ensuring mode consistency in the presence of distributed mode management is challenging. We should not only ensure that the components are put in the appropriate states, but also guarantee that mode managers agree and synchronize before each mode transition. We define and verify the handshake protocol that is implemented for synchronization of mode managers.

The system is implemented in SystemC, which is again used to not only describe the design but also to formally verify the system. The approach presented in this paper extends our earlier work on modeling the centralized mode rich systems. The main novelty of this paper is in the treatment of the distributed mode management problem and proposal of the handshake protocol that ensures synchronization of distributed mode managers.

Paper IV: A Case Study in Modelling a Fault-Tolerant Satellite System through Implementation of Dynamic Reconfiguration via Handshake

To complete long-term missions, the satellite systems should be able to cope with unforeseen adverse conditions, such as improbable simultaneous failures of components in both the main and standby systems. In this paper, we study such a problem and investigate the principles of achieving fault tolerance via dynamic reconfiguration.

The paper presents a case study in modeling and implementation of a generic subsystem of satellites Control and Data Management Unit (CDMU). The architecture of CDMU consists of processor modules, reconfiguration modules, and telemetry modules. If any of the components in both the main and spare CDMU fails, then the system should establish a new communication infrastructure over the healthy components to continue functioning. Such a dynamic reconfiguration can be achieved by introducing a sophisticated handshake protocol between two processor modules as well as the hierarchical Master – Slave coordination.

Since the protocol is complex, we have formally specified it in PROMELA and used SPIN model checking to verify it. Since the protocol introduced over the newly established communication infrastructure is complex and has a large number of different execution paths, verification of correctness by testing would be rather unfeasible. Formal verification by model checking provided us with great support for verification of such a complex fault tolerant system.

Paper V: Towards Systematic Design of Adaptive Fault Tolerant Systems

The complexity of modern large-scale systems requires solutions that ensure that systems autonomously adapt to the operating environment and internal conditions. In this paper, we propose a general pattern for architecting and developing the adaptive fault tolerant systems. The proposed pattern supports a layered design approach that enables separation of concerns and facilitates the structured design of fault tolerance mechanisms. In our representation of the architectural pattern, we define the interfaces between the components at different levels of abstraction to ensure correct propagation of fault tolerance related data.

The high-level coordination of the fault tolerance mechanisms is implemented by an adaptation manager – a component that is responsible for implementing predictive fault tolerance. To specify the adaptation manager, we propose an algorithm that allows the adaptation manager to monitor the state of the system at the run time and implement proactive adaptation. Such an approach ensures that the overall system would continuously deliver the services with the acceptable quality. We believe that the proposed approach ensures a systematic development of adaptive fault tolerant systems.

This work demonstrates how the deployment of the predictive adaptation allows us to ensure that the system would be able to deliver its services with the acceptable quality despite the occurrence of component failures.

Paper VI: A Structured Approach to Architecting Fault Tolerant Services

Service Oriented Architecture (SOA) is a widely used software engineering framework. To enable a rapid service composition, services define their properties in a standard and machine readable format. It enables service discovery, selection, and binding. Service composition introduces the orchestration of the basic services to build applications. However, usually, research on service orchestration focuses on defining the language for service composition that does not support reasoning about such essential features as fault tolerance. Such reasoning can be supported by dependability analysis and architectural modeling.

We demonstrate how to graphically model the architecture of composite services and augment it with various fault tolerance mechanisms. We propose static and dynamic solutions for introducing fault tolerance into the service composition. The structural solutions rely on the availability of redundant service providers that can be requested to provide services in case of failures of the main service providers. This mechanism allows the designers to mask failures of the individual service providers. The dynamic solutions rely on re-execution of failed services to recover from the transient faults of services.

To facilitate the design of complex fault-tolerant services, we introduce a systematic approach to analyzing possible failure modes of services and defining fault tolerance measures. Our approach is inductive and relies on Failure Mode and Effect Analysis. It progressively analyses one component after another in the service execution flow, explores possible fault tolerance alternatives and systematically introduces them into the service architecture.

We believe that our approach supports structured guided reasoning about fault tolerance and enables efficient exploration of the design space.

7 Related Work

In this chapter, we give an overview of related approaches to development and verification of distributed fault-tolerant systems. Development and verification of fault-tolerant distributed systems is an area of active research in these days. To start with, we consider fault tolerance in the context of mode-rich systems. Then, the related work of mode modeling is described. Finally, we discuss the approaches that demonstrate the modeling and verification of fault-tolerant distributed systems in general.

7.1 Modes and Fault Tolerance

In the existing literature, various approaches relying on the concept of modes have been presented. The fault tolerance is a major requirement of the control systems. AGATA (autonomous satellite demonstrator) and formation flying satellites [93] are two innovative projects in the area of space [53] that validate the FDIR strategy in the mode-rich system.

A fault-tolerant control scheme for satellites is presented in [30] that deals with the actuator faults. The authors have proposed an algorithm for error recovery that utilizes a sliding mode control (SMC) method through finite reaching a time in case of actuator faults and external disturbances. This fault tolerant control system can be used for various actuators (i.e., THR and RW). The work presented in [33, 34] have also proposed the SMC approach which is used to reconfigure the operation of control systems to confirm accuracy and robustness of the designed controller. In all cases, authors present the verification of the SMC method through numerical solutions.

Another approach for fault tolerance in satellite's system has been presented in [37]. This work demonstrates the effectiveness of the fault tolerant control with an on-line control allocation method through numerical solutions. The on-line control allocation method shows that faults can be recovered without reconfiguring the controller. In [38, 39], the authors have presented a decentralized system for the accommodation of actuator faults in the case of the faulty satellite by utilizing SMC and fuzzy logic. They have validated their work by modeling of architecture and presenting the numerical solutions confirming the effectiveness of the used approach.

In this thesis, we have proposed an approach for fault tolerance in the mode rich system. In contrast to [30, 34, 37, 38, 39], our approach is not limited to actuators of satellites. It is a generic approach that deals with faults of all units of satellites such as sensors, actuators, and payload instrument. However, we do not provide the numerical solutions for system validation in contrast to above approaches, but we present architectural modeling of the fault tolerance mechanism for the development of the mode-rich system. For verification of the proposed approach, formal verification has been carried out to examine the consistency of a mode logic. The formal verification has been used to verify the correctness of both centralized and distributed mode management.

To ensure synchronization of mode managers controlling mode transitions, we model and verify the handshake procedure that is an essential part of the implementation of dynamic reconfiguration of satellite systems. The reconfiguration process and the handshake procedure enable an efficient handling of errors in complex distributed system. Our approach is different from on-line control allocation [37] as it provides dynamic reconfiguration implementing error handling.

The researcher's work [29, 35] has presented the formal development of the mode-rich AOCS system by refinement in Event B. The authors have explained the concept of mode and mode transitions and have also defined specifications and refinement patterns for development and verification of mode consistency. In particular, the work has focused on the formal development of a layered mode-rich system and formalization of the mode stability property, which is important for the systems with non-instantaneous mode transitions. The authors have verified the consistency of the modes in the layered reconfigurable systems [26] by deriving the mode logic [25] using refinement in Event-B. This approach has aimed at enabling proof-based verification of mode-rich systems modeled in Event- B. Recent development [92] represents a much more complex model containing a large set of invariants describing in detail the relationships not only between modes and phases but also the effect of failures at different stages of communication. The researchers [51, 52, 95] have studied mode-rich systems to examine the problem of mode confusion and automation surprises. They showed retrospective analysis of mode-rich systems to spot the inconsistencies between the mental picture of the mode logic and the actual system mode logic. The approaches [48, 96] utilized theorem proving in PVS, while most of the approaches [51, 52, 95] rely on model-checking.

Our approach focuses on designing fully automatic systems and ensuring their mode consistency. In contrast to [51], our approach emphasizes the complex relationships between the mode logic and system fault tolerance. Unlike [29, 35, 92], in our approach, we relied on design in SystemC and formal verification by model checking in SPIN that is better suited for modeling distributed architectures. Namely, we have designed complex mode-rich systems in SystemC and have verified its correctness via model checking.

7.2 Model-Driven Engineering and Verification of Fault-Tolerant Systems

Let us now discuss the related work in connection to the chosen development frameworks –SystemC, UML, and SPIN Promela.

SystemC provides data types for hardware modeling. In [46, 13], the authors have illustrated that SystemC is a suitable core language for the complex distributed network that builds the system with the highest level of abstraction (functional level) to develop and analyze a model. In [45], the authors have explained the SystemC-based approach for modeling the network-embedded systems. SystemC verification standard provides an application-programming interface for transaction-based verification, constrained and weighted randomization, exception handling, and other verification tasks [20, 21]. In [91], the researchers use a Concurrent and Comparative Simulation (CCS) technique to insert faults in SystemC models to verify the system behavior in the presence of faults.

Model-driven development of complex distributed systems have been discussed in [41, 43, 44]. In [42], UML state chart diagram has been used to model the complex system such as hybrid dynamic systems. Modeling in UML allows us to represent in a compact way the complex interdependencies of mode-rich systems. UML profiles have been obtained by using constraints, stereotypes and tagged values [85]. Stereotypes describe the new types of modified meta-models, constraints define new properties of the proposed model and tagged values present qualified properties of stereotypes. There are various UML profiles to support the development of dependable systems. UML profile for modeling quality of service and fault tolerance characteristics and mechanisms [86] focuses on the design of the complex fault tolerant system that ensures high quality in the analysis model and supports additional functional requirements in the software architecture. Another work [87] has presented fault forecasting by using a specific UML annotation that enhances the system reliability and determines the failure rate of the components. The UML profile for component-based development of dependable systems [88] renders the safety-related requirements in the UML models. The UML profile for dependability analysis of real-time embedded systems [89] has included the dependability requirements in the model definition.

Formal verification is an essential part of ensuring the correctness of complex fault-tolerant distributed systems. In [54], the authors have investigated the use of different techniques for model checking and simulations. In [28], the authors have proposed an abstraction framework based on PROMELA that reduces the complexity of the model checking formal verification by reducing the number of generated states [53]. In [70], the authors have demonstrated how to use a SPIN model checker to verify security properties and propose an approach to translating UML models to PROMELA. Another work focuses on proposing the general rules for translation of UML and SystemC models into SPIN PROMELA [40, 55].

In this thesis, we have used SystemC as a system design language, UML as a graphical modeling language, and SPIN PROMELA as our verification framework. Our work further extends the approaches presented in [85, 86, 87] and demonstrates how to explicitly represent fault tolerance in model-driven development.

Verification of a distributed protocol in SPIN has also been presented in [70]. However, in our work, we deal with a different class of protocols. The use of SPIN also differentiates our work from the research presented in [53].

7.3 Centralized Mode Management in Fault-Tolerant systems

In the centralized mode management, a single node obtains information from the rest of the system and controls the mode transitions of the entire system. In the existing literature, there is a number of approaches that discuss fault tolerance in mode-rich systems. The related work on mode-rich systems with the centralized mode management is presented in this section.

In [49], the mode logic operations such as flight phases and on-board instruments operations for space and avionics systems have been presented and analyzed. The work in [48] presents PVS formal verification to capture the process of formalizing by considering formal specifications of avionics subsystems. The authors have focused on the mode characterization of the real-time environment that tackles the problem of defining requirements for mode transitions [50]. The authors [47] have studied the specifications of a mode-rich system typical in such domains as space and transportation, the notion of the refinement process of these systems, and its realization. These ideas have been further applied for architecting fault tolerance techniques [97]. According to this approach, a mode-centric specification of the system neither defines how the system operates while it is in some specific mode nor how mode transitions occur. It rather imposes restrictions on concrete implementations. Such an approach complements traditional modeling but does not replace it. The authors [50] have described how consistency can be automated in terms of static analysis. The method has been illustrated by verifying small parts of a system after decomposition of functional specifications. This work has been presented using the state chart machine and Traffic Alert and Collision Avoidance System (TCAS) II software. Other researchers [4, 52] have presented a scheme that deals with mode confusion problems by analyzing the system behavior and uses analysis methods that search the models for predictable error forms. Another approach [75] has indicated that several types of faults can be tolerated in degraded modes for extended architecture. The authors have utilized quantitative capability to structure the method. In [63], the authors have presented online failure prediction methods that exhibit less recovery time because of preventive treatment of faults. However, the approaches to proactive fault tolerance are not well integrated into the system development process.

Unlike ours, the work presented in [48] and [49] does not address formal verification. However, we do not use a refinement process, but we structure the mode specifications and requirements using UML. Unlike [97], we have taken an integrated view and analyzed how to combine reasoning about the system mode logic and its functioning. Another method [4, 50] has studied inconsistency of automation due to lack of an error recovery mechanism in the mapping of requirement specifications.

Reliance of UML as a front end of more formal models, allows us to alleviate this problem. Another implemented approach [50] has described the occurrence of mode transition, but the authors have not relied on formal modeling. In contrast to [75], our approach does not work for quantitative analysis of fault tolerance. Therefore, we can extend our proposed work with quantitative aspects such as error rates, etc. However, the proactive fault tolerance presented in [63] is not well integrated into the system development process. As a result, we have addressed this problem by proposing an architectural approach of an adaptive fault tolerant system. The main mechanism of achieving proactive fault tolerance is adaptation [82].

7.4 Fault Tolerance in Distributed Systems

In this section, we overview various methodologies related to the distributed fault tolerant systems.

Virginia Tech [56, 57] has proposed DSACSS that implements many types of distributed control techniques by using Object Oriented Programming in C++. DSACSS has been used to test both developments of centralized and distributed control components. This scheme delivers non-linear compensation for a failed component through mathematical modeling. The dynamically reconfigurable multi-agent systems [59] represent distributed systems in which agents of distributed nodes collaborate to fulfill the system requirements according to the specifications. In this scheme, the authors have proposed a formal approach to check the validity of the multi-agent system using the Event-B method. The authors [60] have also presented the working of distributed environment by incorporating the embedded programming methodology (i.e., Giotto and the object-oriented AOCS Framework) with real-time requirements. The authors have presented the main steps, which are involved in the design of an optimal feedback control in the satellite formation flying controller (i.e., stabilization and optimization of distributed controller). The authors have provided mathematical solutions and computer-based simulations by using Matlab to describe that distributed systems require less bandwidth and less control energy as compared to centralized systems. The authors have implemented a decentralized control system that deals with the tracking of the robots and maintains formation during the transitions [62]. They have provided mathematical solutions for implementing the decentralized control system, but they have not covered the fault tolerance mechanism. The formal development of a distributed system has been presented by researchers along with mathematical modeling [58]. This work explains the program that is executed to fulfill the requirements of the targeted network infrastructure by assuming middleware behavior. The HATS project [81] has provided abstract behavioral specification modeling and verification for analysis of highly configurable adaptive distributed software systems. This technique has been implemented at the system level by using ABS defined tools.

The simulator in [56, 57] has been used to test or verify the system components, but it does not provide architectural modeling of fault tolerance. Although the authors in [59] have described the formal refinement process and sufficient redundancy in case of failures they have not given details for ensuring safety through architectural modeling of reconfiguration or fault tolerance mechanism. In contrast to [58], we have formally modeled and developed the code of fully operational distributed control system. The approach used in [60, 61, 62] does not provide a formal implementation of error recovery, whereas we have proposed a detailed architectural modeling of fault tolerance in distributed systems and also formal verification has been carried out to verify the operation of system components. In contrast to [81], our work enables design space exploration at the early development stages and facilitates explicit representation of mechanisms for proactive fault tolerance.

In [68], the authors propose fault tolerant service architecture [80] by using SOAP for middleware behavior of a service approach named FT-SOAP. It extends the standard WSDL by proposing a new element to describe the replicated web services. An active UDDI mechanism [69] enables an extension of UDDI's invocation API to enable fault-tolerant and dynamic service invocation, but the authors do not provide formal modeling and verification of the fault tolerance mechanism. The authors present composite service architectures by providing a fault tolerance mechanism to develop new distributed applications [74]. The work presented in [64] describes the reliability and availability analysis of service-oriented architectures in the case of possible failures by using BPEL process and formal language SCA-ASM. Web services [65] are comparatively new technology that has obtained wide acceptance as an important implementation of the service-oriented architecture.

In contrast to [68, 69, 74], we have FMEA to facilitate a structured analysis of failure modes.

The integration of FMEA is also discussed in [14, 76, 77]. Other authors [67, 79] demonstrate how to use FMEA for analyzing safety. In [27], a set of generic patterns has been defined for representing formal specifications in Event-B derived from FMEA safety analysis. This work facilitates elicitation of requirements and also supports traceability of safety and fault tolerance requirements into the formal development process in Event-B. A formal approach [67, 71] to introducing fault tolerance to the service architecture by incorporating results of FMEA has been proposed in [72, 73]. In [24], the work focuses on deriving, formalizing and verifying the correctness of the mode transitions in the fault- tolerant control system. The authors have performed FMEA of each particular mode transition to systematically design error-free mode transition scheme. In recent research [94], an FMEA technique is extended with considering all security risks in distributed industrial measurement systems.

Our work is different from [24, 27] because we focus on using FMEA to introduce fault tolerance into the service architecture.

8 Conclusion and Future Work

This chapter outlines the main contributions of the thesis, discusses limitations of the proposed work and indicates the future research directions.

8.1 Conclusions

Nowadays computer-based systems provide a majority of services critical for our society. Therefore, development of the methodologies enabling the design of reliable fault tolerant systems constitutes an important research goal. In our thesis, we have contributed to achieving this goal by proposing a number of approaches facilitating the design of complex fault-tolerant systems.

In our work, we aimed at creating approaches that can be potentially used in industrial practice. Since model-driven development is widely used in modern software engineering, in our thesis, we studied the problem of an explicit representation of fault tolerance in the model-driven systems development. We demonstrated how to represent different aspects of fault tolerance in the UML models.

UML provides us with a convenient graphical notation for representing the static architectural aspects of the system as well as its dynamic behavior. In our work, we have proposed a number of patterns allowing us to introduce fault tolerance into the service architecture, explicitly represent the impact of fault tolerance on the execution dynamics as well as define the communication between system components during error detection, recovery, and reconfiguration.

UML plays an important role in visualizing system requirements, including the requirements introduced by fault tolerance. In our work, we used UML as a graphical front-end to representing the formal models and system design. In particular, we have used it to visualize the requirements of complex mode-rich systems – a large class of distributed fault-tolerant systems that we have studied in our thesis.

The notion of modes constitutes an important mechanism for structuring the behavior of fault-tolerant systems. Typically, the fault-tolerant control systems have a complex mode logic. The correctness of their behavior, as well as overall system reliability, depend on whether the mode transitions are performed in consistent way by all system components. In our thesis, we have proposed a number of architectural and behavioral patterns for design and verification of mode-rich systems. We have demonstrated how to design mode managers – the dedicated components orchestrating mode transitions, structure system reconfiguration required to mask errors and specify the requirements for the forward and backward mode transitions.

In our study of mode-rich systems, we have also proposed the solutions for designing the systems with the distributed mode management. Ensuring consistency of the mode logic in such systems is especially challenging, because the decisions about the mode transitions are made by several independent mode managers. In our work, we have proposed a handshake protocol allowing the mode managers to synchronize the mode transitions. In our thesis, we presented SystemC implementation of several mode-rich systems and demonstrated how to formally verify various aspects of their behavior using SPIN model checker.

Since an error recovery often requires system reconfiguration, in our thesis we have studied the problem of dynamic reconfiguration of control and service-oriented fault tolerant systems. To ensure the reliability of the systems that perform long-term autonomous missions, we have to ensure that the system can cope with unforeseen adverse conditions, such as improbable simultaneous failures. In our thesis, we have proposed an approach that allows a system to recover from such errors by establishing a new communication infrastructure over the healthy components and adopting a specific handshake protocol to resume service provisioning.

In the context of service-oriented fault tolerant systems, we have demonstrated how to structure the system to achieve an architectural plasticity and efficient propagation of fault tolerance related information between the architectural layers. The proposed architectural pattern provides a basis for implementing proactive fault tolerance that ensures a maintenance of a certain quality level in service provisioning.

To facilitate the design of fault tolerant services, we have also proposed a number of patterns allowing to introduce different redundancy and error recovery mechanisms into the service architecture. Moreover, we have demonstrated how to systematically study possible failure modes of services and explore the feasibility of using different error recovery solutions.

We believe that the approaches proposed in this thesis can be potentially applied in the current software engineering practice to facilitate the disciplined and structured development of complex fault-tolerant systems.

Though we believe that our work has made a contribution to the area of model-driven development and verification of fault-tolerant systems, it is not free of several limitations. Firstly, in our work, we have used the generic development frameworks and tools. The absence of a specialized support automating the proposed approach constitutes a limitation of our research. Secondly, in our work, we have focused on modeling logical (qualitative) aspects of fault tolerance. However, the efficiency of introduced fault tolerance measures is usually evaluated by computing system reliability, i.e., using quantitative models. Our work does not provide support for quantitative modeling, which constitutes another limitation of the proposed approach.

8.2 Future Work

The limitations mentioned above lead to possible directions for future work. In the future, it would be interesting to create an integrated engineering environment that automates augmenting the system architecture with various mechanisms for fault tolerance. Moreover, it would also be useful to automate the translation of graphical models to the models used to represent the system design and perform formal verification.

Another interesting future research direction is to define the notion of architectural refinement to facilitate validation of system properties at different architectural layers.

Finally, it would also be interesting to define a mapping between the system models augmented with the proposed patterns for representing fault tolerance and probabilistic models. Such a mapping would enable a quantitative assessment of reliability and facilitate exploration of the system design space.

Bibliography

[1] A.Avizienis, J.-C. Laprie, and B. Randell. Fundamental Concepts of Dependability. In *Proceedings of the 3rd IEEE Information Survivability Workshop*, pages 7- 12, 2000.

[2] 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*, Volume 1, pages 11-33, March 2004.

[3] P.A. Lee, and T. Anderson. Fault Tolerance – Principles and Practice, Prentice Hall, USA, 1990.

[4] L. Leveson, D. Pinnel, S. D. Sandys, S. Koga, and J. Reese. Analyzing Software Specifications for Mode Confusion Potential. In *Proceedings of Workshop on Human Error and System Development*, pages 132-146, 1993.

[5] OBSW formal development in Event B, 2010. http://deploy-eprints.ecs.soton.ac.uk/view/type/rodin=5Farchive.html

[6] Industrial deployment of system engineering methods providing high dependability and productivity 2011. http://www.deploy-project.eu/.

[7] DEPLOY Deliverable D20. Report on Pilot Deployment in the Space Sector.
FP7 ICT DEPLOY Project, 2010. http://www.deploy-project.eu/.

[8] J. R. Wertz. Spacecraft Attitude Determination and Control. Volume 78, 1978.

[9] Y. Zhao, Z. Yang., and J. Xie. Formal semantics of UML state diagram and automatic verification based on Kripke structure. In *Proceedings of the 22nd Canadian Conference on Electrical and Computer Engineering* (CCECE), pages 974-978, 2009.

[10] M. Deubler, M. Meisinger, S. Rittmann, and Kruger, I. Modelling crosscutting services with UML Sequence Diagram. In *Proceedings of the 8th International Conference on Model Driven Engineering* (MoDELS'05), pages 522-536, 2005.

[11] M. Ibrahim, and R. Ahmad. Class Diagram Extraction from Textual Requirements Using Natural Language Processing (NLP) Techniques. In *Proceedings of 2nd International Conference on Computer Research and Development* (ICCRD'10), pages. 200-204, 2010.\

[12] T. Grotker, S. Liao, G. Marin, and S. Swan. System Design with SystemC. Kluwer Academic Publishers, 2002.

[13] IEEE Standard SystemC Language Reference Manual. IEEE standard 1666- 2005, 2006.

[14] Open SystemC Initiative (OSCI): Defining and advancing SystemC standard IEEE 1666–2005. http://www.systemc.org/.

[15] A. Avizienis. Design of Fault-Tolerant Computers. In *Proceedings of the Fall joint Computer* Conference, pages.733-743, USA, 1967.

[16] G. Booch, J. Rumbaugh, and I. Jacobson. The Unified Modelling Language. Massachusetts, USA: Addison-Wesley, 1998.

[17] A UML Profile for MARTE: Modelling and Analysis of Real-Time Embedded systems, Beta 2. 2008. http://www.omg.org/omgmarte/Documents/Specifications/08-06-09.pdf

[18] Open SystemC Initiative (OSCI): Defining and advancing SystemC standard IEEE 1666–2005. http://www.systemc.org/.

[19] M. Fowler. UML distilled: a brief guide to the Standard object modelling language. Boston, Massachusetts, USA: Addison-Wesley, 2003.

[20] L. Singh & L. Drucker. Advanced Verification Techniques: A SystemC Based Approach for Successful Tapeout, 2004.

[21] J-P. Katoen. Concepts, Algorithms and Tools for Model Checking,1999. http://people.cs.aau.dk/~bnielsen/TOV08/material/katoen2.pdf

[22] A. Ebnenasir, R. Hajisheykhi, and S. Kulkarni. Facilitating the Design of Fault Tolerance in Transaction Level SystemC Programs. In *Proceedings of 13th International Conference on Distributed Computing and Networking (ICDCN'12),* pages 91-105, 2012.

[23] M. Cantor. Dr.Dobbs - The World of Software Development. http://drdobbs.com/184415683

[24] L. Laibinis, Y. Prokhorova, E. Troubitsyna, K. Varpaamiemi, and T. Latvala. Derivation and Formal Verification of a Mode Logic for Layered Control Systems. In *Proceedings of the 2011 18th Asia-Pacific Software Engineering Conference (APSEC-11)*, pages. 49-56, IEEE Computer Society, USA, 2011.

[25] Y. Prokhorova, L. Laibinis, E. Troubitsyna, K. Varpaaniemi, and T. Latvala. Deriving a mode logic using failure modes and effects analysis. *International Journal of Critical Computer-Based Systems*, Volume 3, pages 305—328, 2012.

[26] B. Rubel. Patterns for Generating a Layered Architecture. In *Pattern Languages of Program Design*, pages 119-128, Addison-Wesley, 1995.

[27] E. Troubitsyna, I. Lopatkin, Y. Prokhorova, A. Iliasov, and A. Romanovsky. Patterns for Representing FMEA in Formal Specification of Control Systems. In *Proceedings of 13th International Symposium on High Assurance Engineering*, pages 146-151, 2011.

[28] I. Lopatkin, A. Iliasov and A. Romanovsky. Rigorous Development of Dependable Systems using Fault Tolerance Views. In *Proceedings of the 2011 IEEE 22nd International Symposium on Software Reliability Engineering (ISSRE-11)*, pages 180-189, IEEE Computer Society, 2011.

[29] A. Iliasov, E, Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, I. Dubravka, and T. Latvala. Developing Mode-Rich Satellite Software by Refinement in Event B. In *Proceedings of the 15th International Conference on Formal Methods for Industrial Critical Systems (FMICS-10)*, pages 50- 66, Springer-Verlag, Belgium, 2010.

[30] H. Lee and Y. Kim. Fault-tolerant control scheme for satellite attitude control system Theory and Applications, volume 4, pages 1436-1450, 2010.

[31] K. Javed, and E. Troubitsyna. Designing a Fault-Tolerant Satellite System in SystemC. In *Proceedings of the 7th International Conference on Systems (ICONS'12),* pages 49-54, 2012.

[32] A. Tarasyuk, I. Pereverzeva, E. Troubitsyna, T. Latvala, and L. Nummila. Formal Development and Assessment of a Reconfigurable On-board Satellite System, In *Proceedings of 31st International Conference on Computer Safety, Reliability and Security (SAFECOMP 2012)*, LNCS 7612, pages 210-222, Springer, 2012.

[33] M.D.J. Brown, Y. Shtessel, and J. Buffington. Finite Reaching Time Continuous Sliding Mode Control with Enhanced Robustness. 2000.

[34] Y. Shtessel, J. Buffington, and S. Banda. Tailless Aircraft Flight Control Using Multiple Time Scale Reconfigurable Sliding Modes. In *Proceedings of IEEE Transcations on Control Systems Technology*, pages 288-296, IEEE Control system Society, 2002.

[35] A. Iliasov, E, Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, I. Dubravka, P. Vaisanen, and T. Latvala. Verifying Mode Consistency for On-Board Satellite Software. In *Proceedings of 29th International Conference on SAFECOMP*, Volume 6351, pages 126-141, Austria, 2010.

[36] L. Laibinis, E. Troubitsyna, and S. Leppänen. Service-Oriented Development of Fault Tolerant Communicating Systems: Refinement Approach. In *Proceedings of the International Journal on Embedded and Real-Time Communication Systems*, Volume 1, pages 61-85, October 2010.

[37] F. Yan-Ping, C. Yue-Hua, J. Bin, and Y. Ming-kai. Fault Tolerant Control with on-line Control Allocation for Flexible Satellite Attitude Control System. In *P roceeding of the 2nd International Conference in Intelligent Control and Information Processing (ICICIP-11)*, pages 42-46, China, 2011.

[38] S.M. Azizi and K. Khorasani. A Decentralized Cooperative Actuator Fault Accomodation of Formation Flying Satellites in Deep Space. In *Proceedings of 3rd International Systems Conference (SysCon-09)*, pages 230-235, Canada, 2009.

[39] Li. Junquan and K.D. Kumar. Decentralized fuzzy fault tolerant control for multiple satellites attitude synchronization. In *Proceedings of International Conference on Fuzzy Systems*, pages 1836-1843, Taiwan.

[40] L. Ji, J. Ma, and Z. Shan. Research on Model Checking Technology of UML. In *Proceedings of International Conference on Computer Science and Service System*, pages 2337-2340, 2012.

[41] B. Zhang and Y. Chen. Enhancing UML Conceptual Modelling through the use of Virtual Reality. In *Proceedings of the International Conference on System Sciences*, pages 11, 2005.

[42] J. Shyan Lee and P-L. Hsu. UML-Based Modelling and Multi-Threaded Simulation for Hybrid Dynamic Systems. In *Proceedings of the International Conference on Control Sytems*, pages 1207-1212, IEEE Computer Society, 2002.

[43] Q. Chunyan. UML-based software process modelling. In *Proceeding of the International Conference on Computer, Mechanics, Control and Electronic Engineering*, pages 247-250, 2010.

[44] Y. Zhou, Y. Chen, and H. Lu. UML-based Systems Integeration Modelling Technique for the Design and Development of Intelligent Transportation Management System. In *Proceeding of the International Conference in Systems, Man and Cybemetics*, pages 6061-6066, 2004.

[45] F. Fummi, D. Quaglia, and F. Stefanni. A SystemC-based Framework for Modelling and Simulation of Networked Embedded Systems. In *The Forum on Specification and Design Languages*, pages 49-54, 2008.

[46] S.M. Aziz and J.M.D. Tran. Modelling for Performamnce: SystemC Model of A Communication Bus in A Distributed Network. In *Proceedings of the International Conference on Information and Communication Technology*, pages 231-234, 2007.

[47] 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*, pages 601-619, Springer-Verlag, Germany, 2009.

[48] R. W. Butler. An Introduction to Requirements Capture Using PVS: Specification of a Simple Autopilot, May 1996.

[49] S.P. Miller. Specifying the mode logic of a flight guidance system in CoRE and SCR. In *Proceedings of the 2nd workshop on Formal methods in software practice*, pages 44-53, ACM, 1998.

[50] R. Jorge and C. Alfons. Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal, In *Proceedings of the Real Time Systems*, pages 161-197, 2004.

[51] M. Heimdahl and N. Leveson. Completeness and Consistency in Hierarchical State Based Requirements. In *Proceedings of IEEE Transactions on Software Engineering*, pages 363-377, 1996.

[52] J. Rushby. Using model checking to help discover mode confusion and other automation surprises. In *Proceedings of the 3rd Workshop on Human Error, Safety, and System Development*, pages 167-177, 1999.

[53] A.E. Rugina, J_P. Blanquart, and R. Soumagne. Validating failure detection isolation and recovery strategies using timed automata. In *Proceedings of the 12th European Workshop on Dependable Computing*, France, 2009.

[54] Z. Xin-feng, W. Jian-dong, Z. Xin-feng, L. Bin, Z. Jun-wu, and W. Jun. Methods to Tackle State Explosion Problem in Model Checking. In *Proceedings of the 3rd International Symposium on Intelligent Information Technology Application*, pages 329-331, 2009.

[55] A. Habibi and S. Tahar. Design and Verification of SystemC Transaction- Level Models. In *Proceedings of the Design Automation and Test in Europe*, pages 560-565, 2005.

[56] J.L. Schwartz and C.D. Hall. The Distributed Spacecraft Attitude Control System Simulator: Development, Progress, Plans. In *Proceedings of the NASA Space Flight Mechanics Symposium*, 2003.

[57] J.L. Shwartz. The Distributed Spacecraft Attitude Control System Simulator: From Design Concept to Decentralized Control. 2004.

[58] A. Iliasov, L. Laibinis, E. Troubitsyna, and A. Romanovsky. Formal Derivation of a Distributed Program in Event B. In *Proceedings of the 13th International Conference on Formal Methods and Software Engineering*, pages 420-436, 2011.

[59] D. Grostev, A. Iliasov, and A. Romanovsky. Formal Stepwise Development of Scalable and Reliable Multiagent Systems", *Technical report series*, 2010.

[60] T. Brown, A. Pasetti, w. Pree, T.A. Henzinger, and C.M. Kirsch. A reusable and platform-independent framework for distributed control systems. In *Proceedings of the 20th International Conference on Digital Avionics Systems*, pages A1/1 - 6A1/11, 2001.

[61] N. Dadkhah, L. Rodrigues, and A.G. Aghdam. Satellite Formation Flying Controller Design Using an Optimal Decentralized Approach. In *Proceedings of the American Control Conference*, pages 3162-3167, 2007.

[62] J. Lawton, B. Young, and R. Beard. A Decentralized Approach to Elementary Formation Maneuvers. In *Proceedings of the IEEE International conference on robotics and automation*, pages 2728-2733, 2000.

[63] F. Salfner, M. Lenk, and M. Malek. A survey of online failure prediction methods. *ACM Computer Survey.* Volume 42(3), 2010.

[64] R. Mirandola, P. Potena, E. Riccobene, and P. Scandurra. A reliability model for Service Component Architectures. *Journal of Systems and Software. Volume 89,* pages 109-12, 2014.

[65] S. Distefano, C. Ghezzi, S. Guinea, and Raffaela Mirandola. Dependability Assessment of Web Service Orchestrations. *IEEE Transactions on Reliability. Volume 63(3),* pages 689-705, 2014.

[66] D. Bosnacki, and D. Dams, Discrete-Time Promela and Spin. In *Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'98)*, LNCS 1486, pages 307-310, 1998.

[67] K. Sere and E. Troubitsyna. Safety Analysis in Formal Specification. In *Proceeding of FM'99*, LNCS 1709, pages. 1564 – 1583, Springer, 1999.

[68] D. Liang, C. L. Fang, C. Chen, and F. X, Lin. Fault-tolerant web service. In *Proceeding of Tenth Asia-Pacific Software Engineering Conference*, pages 56-61, IEEE Press, 2003.

[69] M. Jeckle, B. Zengler. Active UDDI-An Extension to UDDI for Dynamic and Fault Tolerant Service Invocation. In *Proceeding of 2nd International Workshop on Web and Databases*, pages 91-99, Springer 2002.

[70] P. S. Kaliappan and H. Koenig. Designing and Verifying Communication Protocols Using Model Driven Architecture and Spin Model Checker. In *Proceedings of the International Conference on Computer Science and Software Engineering*, pages 227-230, 2008.

[71] E. Troubitsyna. Elicitation and specification of safety requirements. In *Proceeding of Third International Conference on Systems (ICONS 08)*, pages 202-207, 2008.

[72] L. Laibinis, E. Troubitsyna, and S. Leppänen. Service-Oriented Development of Fault Tolerant Communicating Systems: Refinement Approach. In *Proceeding of International Journal on Embedded and Real- Time Communication Systems*. Volume 1, pages 61-85, 2010.

[73] L. Laibinis, E. Troubitsyna, A. Iliasov, and A. Romanovsky. Rigorous development of fault-tolerant agent systems. In *Proceeding of Rigorous Development of Complex Fault-Tolerant Systems*, pages 241-260, Springer 2006.

[74] V. Dialani, S. Miles, L.Moreau, D. Roure, and M. Dialani. Transparent fault tolerance for web services based architectures. In *Proceeding of 8th Europar Conference (EULRO-PAR02)*, pages 889-898, 2002.

[75] J. Lygeros, D. N. Godbole, and Broucke. M.E.: Design of an extended architecture for degraded modes of operation of ivhs. In: In *American Control Conference*, pages 3592–3596, 1995.

[76] FMEA Info Centre. http://www.fmeainfocentre.com/.

[77] N.G. Leveson. Safeware: system safety and computers. Addison-Wesley, 1995.

[78] N. Storey. Safety-critical computer systems. Addison-Wesley, 1996.

[79] F. Ortmeier, M. Guedemann, and W. Reif. Formal failure models. In *Proceedings of the IFAC Workshop on Dependable Control of Discrete Systems (DCDS 07)*, Elsevier, 2007.

[80] Web Services Architecture Requirements. http://www.w3.org/TR/wsa-reqs/.

[81] R. Hahnle. HATS Project: Highly Adaptable and Trustworthy Software using formal models. In *Proceedings of the 4th International Symposium on Leveraging Applications (ISoLA'10)*, pages 3-8, 2010.

[82] L. Lao, M. Ellis, and P. D. Christofides. Proactive fault-tolerant model predictive control: Concept and application. In *Proceedings of the American Control Conference (ACC'13),* pages 17-19, 2013.

[83] L. Laibinis and E.Troubitsyna. Fault tolerance in a layered architecture: a general specification pattern in B. In *Proceeding of SEFM 2004*, pages 346- 355, IEEE Computer Press, 2004.

[84] E. Troubitsyna. Failure Modes and Effect Analysis of Use Cases: A Structured Approach to Engineering Fault Tolerance Requirements. In *Proceedings of the 4th International Conference in Dependability (DEPEND'11)*, pages 82-87, 2011.

[85] OMG. UML 2.0 Superstructure Specification, 2005. http://doc.omg.org/formal/2005-07-04.pdf

[86] OMG. UML Profile for Modelling Quality of Service (QoS) and Fault Tolerance Characteristics and Mechanisms, 2006. http://www.omg.org/spec/QFTP/1.0/PDF/

[87] V. Cortellessa and A. Pompei. Towards a UML profile for QoS: a contribution in the reliability domain. *SIGSOFT Software Engineering Notes*. Volume 29, pages 197-206, 2004.

[88] J. Jurjens and S. Wagner. Component-based Development of Dependable Systems with UML. In *Component-Based Software Development for Embedded Systems – An Overview on Current Research Trends*. Volume 3778 of LNCS, pages 320-344. Springer-Verlag, 2005.

[89] S. Bernardi and J. Merseguer. A UML profile for dependability analysis of real-time embedded systems, In *P roceedings of the 6th international workshop on Software and performance (WOSP'07)*, pages 115-124, ACM Press, 2007.

[90] L. Laibinis, E. Troubitsyna, S. Leppänen, J. Lilius, and Q. Malik. Formal Service-Oriented Development of Fault Tolerant Communicating Systems. In *Proceedings of the Rigorous Development of complex Fault-Tolerant Systems*, LNCS 4157, pages 261-287, Springer 2006.

[91] J. C. Laprie. Dependability: Basic Concepts and Terminology. Springer- Verlag, 1991.

[92] Lu. Weiyun and M. Radetzki. Efficient Fault Simulation of SystemC Designs. In *Proceeding of the 14th Euromicro Conference on Digital System Design (DSD)*, pages 487–494, 2011.

[93] A. Tarasyuk, I. Pereverzeva1, E. Troubitsyna1, and T. Latvala. The Formal Derivation of Mode Logic for Autonomous Satellite Flight Formation. In *Proceedings of the 34th International Conference on Computer Safety, Reliability, and Security (SAFECOMP'15)*, pages 29-43, 2015.

[94] C. Schmittner, T. Gruber, P. Puschner, and E. Schoitsch. Security Application of Failure Mode and Effect Analysis (FMEA). In *Computer Safety, Reliability, and Security.* Volume 8666, pages 310-325, 2015.

[95] B. Buth. Analysing mode confusion: An approach using fdr2. In *Proceedings of SAFECOMP*, pages 101–114. Springer, Lecture Notes in Computer Science, Vol. 3219, 2004.

[96] A. Joshi, S. P. Miller, and M. P. Heimdahl. Mode confusion analysis of a flight guidance system using formal methods. In *Proceedings of 22nd IEEE Digital Avionics Systems Conference (DASC'2003)*, Indianapolis, USA, October 2003.

[97] I. Lopatkin, A. Iliasov, and A. Romanovsky. On fault tolerance reuse during refinement. In *Proceedings of 2nd International Workshop on Software Engineering for Resilient Systems*, April 2010.

[98] A. Pnueli. The temporal semantics of concurrent programs. In *18^{th} Annual Symposium on Foundations of Computer Science*, 1977.

[99] S. Owicki and D. Gries. *Verifying properties of parallel programs: an axiomatic approach.* Volume 19, number 5, pages 279-285, ACM, 1976.

[100] S. Owicki and L. Lamport. *Proving liveness properties of concurrent programs*. Volume 4, Number 3, pages 455-495, ACM, 1982.

[101] E. A. Emerson. *The Beginning of Model Checking: A Personal Perspective*. Pages 27-45, Springer-Verlag, 2008.

[102] M. C. Edmund. *The Birth of Model Checking*. Pages1-26, 2008.

Part II

Original Publications

Paper I.

Kashif Javed, Asifa Kashif, and Elena Troubitsyna, "Implementation of SPIN Model Checker for Formal Verification of Distance Vector Routing Protocol", International Journal of Computer Science and Information Security (IJCSIS), Vol 8, No 3, pp. 1-6, June 2010, USA, ISSN 1947-5500.

Implementation of SPIN Model Checker for Formal Verification of Distance Vector Routing Protocol

KASHIF JAVED
Department of Information
Technologies Abo Akademi University
Turku, FIN-20520, Finland
Kashif.javed@abo.fi

ASIFA KASHIF
Department of Electrical Engineering
National University of Computer and Emerging Sciences, Islamabad, Pakistan
asifa.ilyas85@gmail.com

ELENA TROUBITSYNA
Department of Information
Technologies Abo Akademi University
Turku, FIN-20520, Finland
Elena.Troubitsyna@abo.fi

Abstract—**Distributed systems and computing requires routing protocols to meet a wide variety of requirements of a large number of users in heterogeneous networks. DVR is one of many other employed protocols for establishing communication using routes with minimum cost to different destinations from a given source. Research work presented in this paper focuses on implementation of DVR in SPIN and provides formal verification of correctness of DVR behaviour covering all required aspects. Simulation results clearly show a proof of the established paths from each router to different destinations in a network consisting of six routers and a number of links.**

Keywords: Formal Verification, DVR Protocol, SPIN Model Checker, Distance Vector Routing, Implementation in PROMELA

I. INTRODUCTION

A computer network consists of a number of routers which have the capability to communicate with each other. Routing Information Protocol (RIP) is widely used for routing packets from a source to its destination in computer networks. RIP requires information about distance and direction from source to destination. Each router, in the Distance Vector Routing (DVR) methodology, keeps updated record of distances and hops of its neighbours. Various techniques are used to gather useful routing table information for each router. In one approach, special packets are sent by each router and are received back after having time-stamped by the receivers. Chromosomes have been employed in the Genetic Algorithm [1] to select the most optimal path by utilizing its fitness function, selection of next generation and crossover operation for updating the routing tables in an efficient manner. Thus, all routers keep refreshing their routing tables and maintain latest information about other neighbouring routers in order to provide optimized performance in the available network [1-3].

Mahlknecht, Madni and Roetzer [4] has presented an efficient protocol that uses hop count and cost information in its Energy Aware Distance Vector (EADV) routing scheme and makes use of shot-multi-hop routing for consuming lesser energy in the wireless sensor networks. EADV can do well for long lasting battery-powered sensor nodes while using the lowest cost path towards the selected sink node. An algorithm is considered the most effective if it contains the correct and latest information about its neighbours in its DVR table. An effort has been made by Liwen He by devising a computational method to protect a network from internal attacks (such as mis-configuration and compromise) through the use of verifying routing messages in the DVR protocols [5]. Formal verification of standards for DVR protocols has also been comprehensively presented by Bhargavan, Gunter and Obradovic [6] using three case studies. The researchers have used HOL (an interactive theorem prover and SPIN (model checker) to verify and prove salient properties of DVR protocols. HOL and SPIN have been employed by these researchers for providing a proof of convergence for the RIP [7].

The remaining paper is organized as follows. DVR protocol is presented in Section II and Section III describes the use of SPIN tool and PROMELA language for formal verification. System design and implementation has been discussed in Section IV covering network topology, implementation details and operation of DVR protocol. Formal verification of simulation results has been illustrated in Section V and finally conclusions and future work is given in Section VII.

II. DISTANCE VECTOR ROUTING PROTOCOL

A. *General Methodology*

A routing table is required to be maintained for each router in the network for the purpose of working of a DVR scheme. Routing table information is used to determine the best path (i.e. having minimum cost in terms of distance or hops) from a source to destination. Links are needed to connect concerned routers for establishing communication. An optimal DVR protocol has to exchange frequent messages in order to update the routing table of each router. So, exchanging information among neighbours is carried out on regular intervals.

Routing table of every router keeps necessary information (i.e. id of neighbouring routers, most suitable outgoing link to be used for the destination, distance, hops (number of routers on the route), time delay, number of queued messages on the link). The process of making forwarding decision for selecting the best optimal path from source to destination is based on a combination of these parameters. The objective of routers is to send packets to hosts connected to the networks for heterogeneous requirements of a large number of users. In this way, efficient DVR schemes ultimately establish good global paths by connecting hosts in a distributed environment covering very long distances. Those routers are taken as neighbours which have links/interfaces to a common network.

B. *Routing Information Protocol*

RIP [8,9] is a widely used protocol for finding the optimal path to the destination in a network. Each router has a routing table and all routers periodically updated their routing tables by using advertising approach. All routes of a router are advertised through the mechanism of broadcasting RIP packets to all the neighbouring routers in the network. Every router checks the advertised information of neighbouring nodes and changes information only in its routing table if the new route to the same destination further improves the existing route length. In other words, the updated routing table information now takes to the best available route so far for the relevant destination.

The number of hops in the RIP are kept low (up to 15) for the route length for faster convergence [6,7]. RIP methodology, however, prevents formation of loops between pairs of routers in order to minimize convergence time as well as permitted route length. Timer expiry record is also maintained in every routing table and is normally set to 180 seconds whenever a routing table is updated. As routers advertise after every 30 seconds, the destination is considered unreachable if a router is not refreshed for 180 seconds. It further waits for another 120 seconds. If the router remains un-refreshed during this time as well, then its route is removed from the routing tables of the concerned routers. This requirement is incorporated to cater for broken links, faulty networks and congestions.

III. USE OF SPIN AND PROMELA

A. *Formal Verification*

A number of new systems and methodologies are being devised by the researchers in different areas of science, technology and engineering as a result of meaningful R&D work being undertaken by academic and research institutes all over the world. Every proposed system requires a proof of its correctness by gathering results using simulation and testing techniques. Formal verification terminology [10,11] is in fact a process of actual demonstration of the system in order to check its correctness under the defined boundaries and valid conditions of used parameters/variables.

Precision and accuracy of the system is verified by running the programming modules by employing required algorithms in the model checking approach. Errors occurred (if any) are properly identified under varying conditions so that such errors can be easily located by the users and are later on repaired/tackled by adjusting specifications of the model. Afterwards, the model description is fine tuned to achieve required model specifications for verification of correct results of the system.

B. *SPIN Tool and PROMELA High Level Language*

SPIN [12,13] is a open-source software tool and is widely used for the formal verification of software systems working in the distributed environment. Inspiring applications of SPIN include the verification of the control algorithms for various applications, logic verification of the call processing software for a commercial data communication, critical algorithms for space missions, operating systems, switching systems, distributed & parallel systems and formal verification of various routing protocols. This tool also supports interactive, random and guided simulations for a wide variety of applications. Spin can be used in four main modes (i.e. as a simulator, as an exhaustive verifier, as a proof approximation system and as a driver for swarm verification).

Spin provides efficient software verification and supports the PROMELA (PROcess MEta LAnguage) high level language to specify systems descriptions [14]. It is a SPIN's input language which is used to build detailed PROMELA models for complete verification of system designs. It provides a way for making abstractions of distributed systems. Different assumptions are used in SPIN to verify each model. After checking correctness of a model with SPIN, it can then be used to build and verify subsequent models of the system so that the fully developed system produces the required behavior. PROMELA programs consist of processes, message channels, and variables.

IV. SYSTEM DESIGN AND IMPLEMENTATION

A. *Network Topology*

The network topology shown in Figure 1 has been used for implementation of DVR protocol. There are six routers (A, B, C, D, E & F) and seven links (edges). Each link connects two routers. Weight values range from 2 to 23 for different links and these values indicate distances between routers. Integer values have been used and distance units can be chosen during actual implementation of the network. For example, the distance between routers A and C via B is 6 using 2 hops and via D, E and F is 33 using 4 hops.

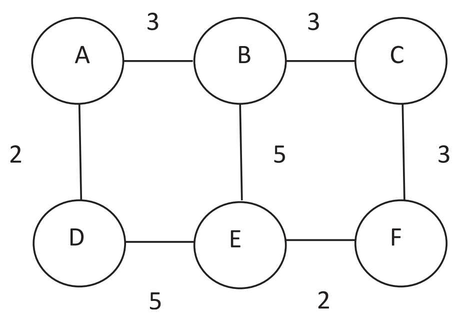


Figure 1: Network Topology

B. *System Implementation*

SPIN's PROMELA language has been used to construct complete model of DVR protocol on a Pentium machine. Packets from the source to destination travel using links provided by routers by making use of their routing tables for the given distributed environment of the network. After

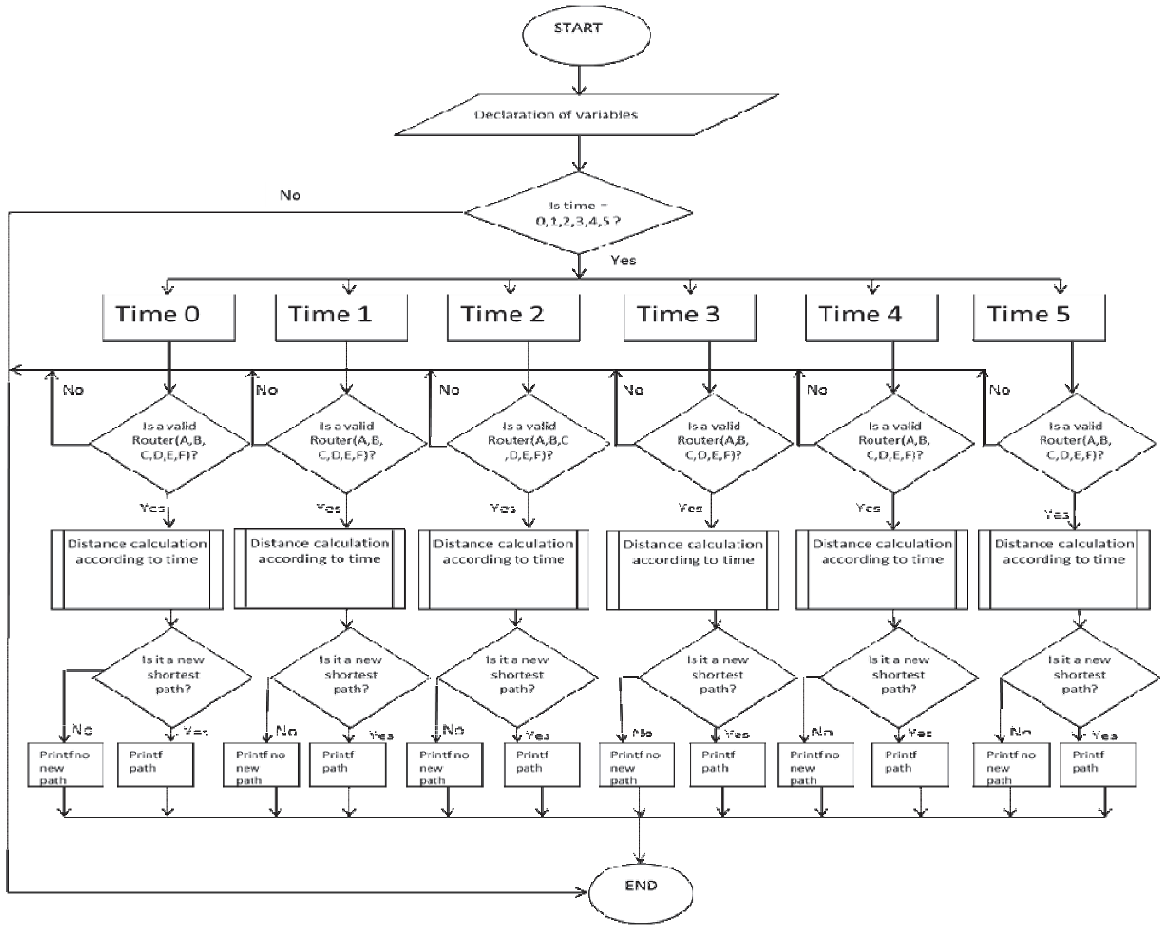


Figure 2: System Flowchart

initialization of the variables, distance is calculated from each router at time period T=0, T=1, T=2, T=3, T=4 and T=5. At each stage it checks whether the measured distance forms a new shortest path or not. Whenever the shortest path is found from the source to destination, routing table entry for the concerned router is automatically updated to make good forwarding decision in order to ensure optimal path, having minimum distance, for faster communication. Thus, each router updates its routing table after each time period. The main objective of the DVR protocol is to provide the current best route (path) from source to destination for each communication. Flowchart of the modeled system in SPIN/PROMELA is shown in Figure 2.

For the given network, the PROMELA program has six processes (one for each time period) to find distance based upon the time period conditions (0 to 5). The found distance from a particular source to destination for each time period is compared with all the available alternate routes. Router's table is only updated if the new distance is minimum between the selected source and destination. The new shortest path is recorded after each calculation. If the determined route does not find minimum distance during the given time period, then it ignores its path without updating any entry in the routing table. Routers improve their routes whenever a router advertises its routing table to its neighbours. So, new routes are determined purely based on their length measured in distance. For timely convergence, the number of hops involved in the length is limited to 15 as already highlighted by Bhargavan et. al. [7].

C. Operation of DVR Protocol

DVR protocol works independently for every destination and it is assumed that there is no topology change for protocol's convergence during every time period. The router broadcasts after every 30 seconds and the destination is taken as inaccessible if it is not refreshed for 180 seconds. The route is removed from the tables of concerned routers if the particular router fails to refresh itself for 300 seconds.

		Via							Via							Via					
	From A	**A**	**B**	**C**	**D**	**E**	**F**	**From B**	**A**	**B**	**C**	**D**	**E**	**F**	**From C**	**A**	**B**	**C**	**D**	**E**	**F**
T=0	**A**							**A**	3						**A**						
	B		3					**B**							**B**		3				
	C							**C**			3				**C**						
	D				23			**D**							**D**						
	E							**E**					5		**E**						
	F							**F**							**F**						3
T=1	**A**							**A**							**A**		6				
	B							**B**							**B**						
	C		6					**C**							**C**						
	D							**D**	26				10		**D**						
	E		8		28			**E**							**E**		8				5
	F							**F**			6		7		**F**						
T=2	**A**							**A**					33		**A**						
	B				33			**B**							**B**						10
	C							**C**					10		**C**						
	D		13					**D**							**D**		13				10
	E							**E**	31		8				**E**						
	F		9		30			**F**							**F**		10				
T=3	**A**							**A**							**A**		36				13
	B							**B**							**B**						
	C		13		33			**C**							**C**						
	D							**D**			13				**D**						
	E		11					**E**							**E**		34				
	F							**F**	33						**F**						
T=4	**A**							**A**			36				**A**						
	B				36			**B**							**B**						36
	C							**C**	36						**C**						
	D		16					**D**							**D**						36
	E							**E**							**E**						
	F				39			**F**							**F**		36				
T=5	**A**							**A**							**A**						
	B							**B**							**B**						
	C							**C**							**C**						
	D							**D**							**D**						
	E							**E**							**E**						
	F							**F**							**F**						

Table1: Calculated Distance from Routers A, B and C for Different Destinations at Time Periods T=0 to T=5

Although the PROMELA's built model can be used for any number of routers but its operation is restricted only to the topology given in Figure 1. For the purpose of explanation of the model, it is assumed that every router operates without any problem and updates its routing table during regular intervals of time.

At Time=0, it calculates distances to neighbouring routers from each router having maximum one hop. Thus, distance from A to B is 3 & A to D is 23from router A; from router B it is 3, 3 & 5 for routers A, C & E respectively; and distances are 5, 5 & 2 for routers B, D & F respectively from router E. These distances can be observed in Tables 1 and 2. Now two hops from the current router are taken for T=1. So, distance from A to C via B is 6; A to E via D 28; and A to E via B is 8 as given in Table 1.

When T is taken as T=2, three hop lengths are counted for determining the distance from each router. From router D, measured distances are 13 via E to A, 29 via A to C, 10 via E to C and 31 via A to E. Same can be seen in Table 2. Hop length is four when T=3, distance covered to B via E, D via C and D via E is 33, 16 and 33 respectively from router F as shown in Table 2. Similarly, routes have distances of 36 (B via F), 36 (D via F) and 36 (F via B) from router C (for T=4) as given in Table 1. Both Tables 1 and 2 clearly indicate that no routes are available from any router when T=5 (six hops) for network configuration of Figure 1.

		Via							Via							Via					
	From D	**A**	**B**	**C**	**D**	**E**	**F**	**From E**	**A**	**B**	**C**	**D**	**E**	**F**	**From F**	**A**	**B**	**C**	**D**	**E**	**F**
T=0	**A**	23						**A**							**A**						
	B							**B**		5					**B**						
	C							**C**							**C**			3			
	D							**D**				5			**D**						
	E					5		**E**							**E**					2	
	F							**F**						2	**F**						
T=1	**A**							**A**		8		28			**A**						
	B	26				10		**B**							**B**			6		7	
	C							**C**		8				5	**C**						
	D							**D**							**D**					7	
	E							**E**							**E**						
	F					7		**F**							**F**						
T=2	**A**					13		**A**							**A**			9		10	
	B							**B**				31		8	**B**						
	C	29				10		**C**							**C**					10	
	D							**D**		31					**D**						
	E	31						**E**							**E**			11			
	F							**F**		11					**F**						
T=3	**A**							**A**						11	**A**						
	B					13		**B**							**B**					33	
	C							**C**				34			**C**						
	D							**D**							**D**			16		33	
	E							**E**							**E**						
	F	32				16		**F**							**F**						
T=4	**A**					16		**A**							**A**			39			
	B							**B**							**B**						
	C	36						**C**							**C**					36	
	D							**D**						34	**D**						
	E	34						**E**							**E**			37			
	F							**F**				37			**F**						
T=5	**A**							**A**							**A**						
	B							**B**							**B**						
	C							**C**							**C**						
	D							**D**							**D**						
	E							**E**							**E**						
	F							**F**							**F**						

Table 2: Calculated Distance from Routers D, E and F for Different Destinations at Time Periods T=0 to T=5

V. Formal Verification of Simulation Results

The implemented system in PROMELA programming language has been tested exhaustively and obtained simulation results are shown in Tables 1 and 2. Spin model checker has been used to verify all the results. The developed model ensures that all the routers correctly maintain and update their tables as and when new routes are searched and visited. The broadcast mechanism works well at different time periods and the system provides correct and optimized results from each router to various destinations depending upon network topology, layout of routers and links connecting different routers in the network.

The SPIN's verification model successfully checks all the available routes via different routers and permits only the shortest path from the available options. It is evident from the following decisions (only four out of many are presented here):

1) *At T =1, the route length from E to C via B is 8 where as it is 5 via F. So, E router adopts F router's path to reach C.*
2) *The distance between routers B & E via A and via C is 31 and 8 respectively. SPIN's checker confirms that minimum distance is covered for reaching to C from E when T=2.*
3) *When T=3, the path cost determined by the model is 13 from C to A via F, E & B but another path for connecting the same two router via B, E & D is 36,*

each path makes use of four hops. Of course, the longer path is simply ignored.

4) *Similarly, route length from F to D through C, B & A is 32 and it is 16 via routers C, B & E. A saving of 16 is noted while using the most economical path.*

A careful analysis of the simulation results shown in Tables 1 & 2 clearly indicates that the modeled system in PROMELA operates correctly and provides the best possible routes involving minimum distances using DVR protocol on the given network environment. The system works efficiently under all conditions and the SPIN model checker has guarantees correctness of all results. It means that all the routing tables are timely updated while messages are being sent to various destinations from a particular source. Now, this can be extended to bigger networks in the distributed environment for efficient and correct functioning using SPIN tool.

VI. CONCLUSIONS AND FUTURE WORK

Many researchers have implemented DVR protocols for various applications. In this research work, PROMELA language has been used to implement DVR protocol on a six router model. Formal verification of DVR protocol properties has been shown through the use of SPIN checker model. The simulation results amply demonstrate correctness and reliability of DVR protocol under varying conditions. Performance of the implemented has been extremely well and it can further be improved to make it more efficient in terms of reducing storage space requirements, incorporating security mechanism for safer communication, minimizing congestion at peak loads and making it fault-tolerant for enhancing its reliability and flexibility.

REFERENCES

[1] M. R. Masillamani, A. V. Suriyakumar, R. Ponnurangam and and G.V.Uma, "Genetic Algorithm for Distance Vector Routing technique", AIML International Conference, 13-15 June 2206, Egypt, pp. 160-163.

[2] Andrew S.Tanenbaum, "Computer Networks", 4th Edition,. Prentice-Hall Inc., 2005.

[3] G. Coulouris, J. Dollimore and T. Kindberg, "Distributed Systems : Concepts and Design, 4th Edition, Addison-Wesley, 2005.

[4] S. Mahlknecht, S. Madani and M. Rötzer, "Energy Aware Distance Vector Routing Scheme for Data Centric Low Power Wireless Sensor Networks," *Proceedings of the IEEE International Conference on Industrial Informatics INDIN 06*, Singapore, 2006.

[5] Liwen He, "A Verified Distance Vector Routing Protocol for Protection of Internet Protocol", Lecture Notes in Computer Science, Networking – ICN 2005, Volume 3421, Springer, pp. 463-470.

[6] K. Bhargavan, D. Obradovic and C. A. Gunter, "Formal Verification of Standards for Distance Vector Routing Protocols", Journal of the ACM, Vol. 49, no. 4, July 2002, pp. 538-576.

[7] K. Bhargavan, C. A. Gunter, and D. Obradovic, "Routing Information Protocol in HOL/SPIN", Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics 2000, August 14 - 18, 2000, London, UK, pp. 53-72.

[8] C. Hendrick, "Routing Information Protocol", RFC 1058, IETF, June 1988.

[9] G. Malkin, 'RIP Version Carrying Additional Information', IETF RFC 1388, January 1993.

[10] J. Katoen, "Concepts, Algorithms and Tools for Model Checking", Lecture Notes 1998/1999, Chapter1: System Validation.

[11] N. A. S. A. Larc, "What is Formal Methods?", http://shemesh.larc.nasa.gov/fm/fm-what.html, formal methods program.

[12] R. de Renesse and A. H. Aghvami "Formal Verification of Ad-Hoc Routing Protocols using SPIN Model Checker", Proceedings of IEEE MELECON'04, Croatia, May 2004.

[13] G. J. Holzmann, "The Model Checker SPIN", IEEE Transactions on Software Engineering, Vol. 23, No. 5, May 1997, pp. 279-295.

[14] G. J. Holzmann, "Design and Validation of Computer Protocols", Prentice Hall, November 1990.

Paper II.

Kashif Javed, Elena Troubitsyna, “Designing a Fault-Tolerant Satellite System in SystemC”, ICONS 2012, The Seventh International Conference on Systems, pp. 49–54, IEEE Computer Press, March 2012, Saint Gilles, Reunion Island.

Designing a Fault-Tolerant Satellite System in SystemC

Kashif Javed
Department of Information Technologies
Abo Akademi University
Turku, FIN-20520, Finland
Kashif.Javed@abo.fi

Elena Troubitsyna
Department of Information Technologies
Abo Akademi University
Turku, FIN-20520, Finland
Elena.Troubitsyna@abo.fi

Abstract—**Designing fault-tolerant satellite systems is a challenging engineering task. Often behavior of satellite systems is structured using notion of modes. Ensuring correctness of mode transitions is vital for guaranteeing safe and fault-tolerant functioning of a satellite. In this paper, we propose an approach to designing fault-tolerant satellite systems in SystemC. We demonstrate how to develop Attitude and Orbit Control System in SystemC and verify its correctness via model checking.**

Keywords-component; Fault-Tolerance; Mode-Rich Systems; Design; Verification

I. INTRODUCTION

Designing a system controlling a spacecraft is a challenging engineering task. The system should satisfy a large number of diverse functional and non-functional requirements. In particular, the designers should aim at building a fault-tolerant system, i.e., the system that should cope with faults of various system components. Often behavior of satellite systems is structured using the notion of modes – mutually exclusive sets of system behavior. Fault-tolerance is achieved by putting the system to some downgraded mode when an error occurs. In this paper, we consider an Attitude and Orbit Control System (AOCS) – a generic subsystem of a spacecraft [1]. We demonstrate how to achieve fault-tolerance via backward mode transitions.

AOCS is a complex control system consisting of several components. To ensure correctness of mode transition, we need to guarantee that all components reach a certain state. Moreover, when a component fails we need to guarantee that all other components make an appropriate backward transition.

In this paper, we propose an approach for designing more-rich system in SystemC programming language. We propose an algorithm defining mode-transition scheme of AOCS. To confirm correctness of our algorithm, we have converted it into Promela [6,7] and the results have been verified using SPIN model checker [7,8].

Section II presents architecture of the system. Unit branch state and state transitions have been explained in Section III and the controller phases & phase transitions of the AOCS are described in Section IV. Mode transitions and fault-tolerance procedures for correct functioning of the satellite under faulty conditions are illustrated in Sections V and VI respectively. Section VII explains verification of the implemented system and the paper is summarized in Section VIII besides giving direction for the future work.

II. ARCHITECTURE

The main purpose of AOCS is to control attitude and orbit [1] of a satellite. AOCS consists of a number of components -- AOCS Manager, FDIR (Failure Detection, Isolation and Recovery) Manager, Mode Manager and Unit Manager. The AOCS manager plays key role while dealing with the processing of sensor data, managing actuator movements relating to the units of Reaction Wheel (RW) and Thruster (THR) and doing computation for various controls. The responsibility of FDIR is to timely deal with such tasks as failure detection, isolation and recovery. Mode transitions are handled by the Mode Manager whereas the Unit Manager deals with unit reconfigurations and unit level state transitions [2,3]. Mode and Unit Manager Architectures are further elaborated in the following paragraphs.

A. Mode Manager

The responsibilities of mode include checking of mode transition preconditions, execution of mode transitions, management of controller phases and partially management of related units. There are six different types of controlled modes (i.e. Off, Standby, Safe, Nominal, Preparation and Science) in the mode manager and each mode has its own well-defined unique function. A brief summary of these modes is given below:

1) Off Mode: The satellite is immediately switched in the off mode as soon as the AOCS software booting is completed from the central data management unit.

2) Standby Mode: It is important to check and ensure successful separation of the spacecraft from the launcher and this work is continuously monitored and completed by the software process during the standby mode.

3) Safe Mode: Satellite enters this mode when the separation from the launcher is done. As soon as the system is in the safe mode, the relevant portions of Earth Sensor (ES), RW (Reaction Wheel) and Sun Sensor (SS) are switched to on state, the coarse pointing controller goes in the running phase and fine pointing controller is put in the idle phase. Initially the satellite acquires a stable attitude and then it achieves the coarse pointing.

4) Nominal Mode: When a mode transitions to nominal, the coarse pointing controller becomes idle and the fine pointing controller is set to the running phase. The selected branches of RW, Star Tracker (STR) and THR are switched to on state. In this mode, the satellite utilizes fine pointing control so that the Payload Instrument (PLI) in the AOCS is properly used for measurements.

5) Preparation Mode: The moment the mode is transitioned to the preparation, the concerned portion of Global Positioning System (GPS) is set to fine state, the relevant branch of PLI is switched to standby state and needed processes of RW, STR and THR go to on state. Thus, this mode ensures that the fine pointing control is reached and PLI gets ready for fulfilling its required tasks.

6) Science Mode: In science mode, the selected branch of GPS remains in the fine state, the concerned branch of PLI goes in the science state and the relevant parts of RW, STR and THR maintain their on state. Therefore, the PLI in this mode is ready to perform the tasks for which it has been designed. It stays in this mode till the completion of planned tasks.

B. Unit Manager

The AOCS consists of seven different units and internal state changes in these units are controlled by the unit manager. Mode manager controls the components of unit manager. Seven different controlled units are ES, SS, STR, GPS, RW, THR and PLI. Their brief description is as under:

1) ES is a device that measures the direction to the earth in the sensor's field of view. ES's internal state is either on and off.

2) SS is a tool to measure the direction to the sun in the sensor's field of view. It is also in the on or off state.

3) STR is an optical device that measures the position of stars in its field of view and performs pattern recognition on these stars in order to identify the portion of the sky at which it is looking. Two possible STR's operational states are on and off.

4) GPS is a sophisticated gadget that receives readings related to the satellite position and makes calculations to determine satellite's attitude. Two possible states of GPS operation are coarse navigation and fine navigation.

5) RW is a rotating wheel which is essentially required in order to apply the required torque to the satellite. It is achieved by accelerating or breaking the wheel. RW's state can be either on or off.

6) THR is a position actuator that is used to force the satellite to change its position and its orbit by emitting gas. It can also be in either on or off state.

7) The PLI is an instrument which provides required measurements pertaining to the specific mission. It can operate in standby or science state.

III. UNIT BRANCH STATE AND STATE TRANSITIONS

Every unit is implemented as a pair of identical devices to maintain the nominal branch and the redundant branch. For each unit, one and only one branch is selected at a time. Every selected branch is in on state and its status is locked. In other words, a branch in the off state is always allocated an unlocked status.

In total, there are six states of unit components (i.e. on, off, coarse, fine, standby and science). Whenever an unit state goes from off to on, the powering takes place. Similarly, when the unit switches from on to off state, un-powering takes place. Powering and un-Powering are associated with the states and state transitions of a branch of ES, SS, STR, RW or THR. Occurrence of such states and state transitions is shown in Figure 1. For the GPS unit, unit state goes from off to coarse state and coarse to fine state, then powering and upgrading is carried out respectively. In case of fine to off state transition, first downgrading is performed then un-powering is done. States and State Transitions of a Branch of GPS are depicted in Figure 2.

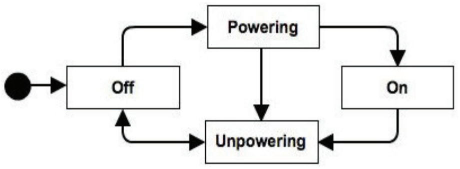


Figure 1: States and State Transitions of a Branch of ES, SS, RW, STR or THR [1]

In case of PLI unit, when the unit state goes from off to standby and from standby to science state, then powering and upgrading is achieved respectively. In case of science to off state transition, first downgrading occurs and then un-powering takes place. Figure 3 demonstrates states and their transitions of a branch of PLI.

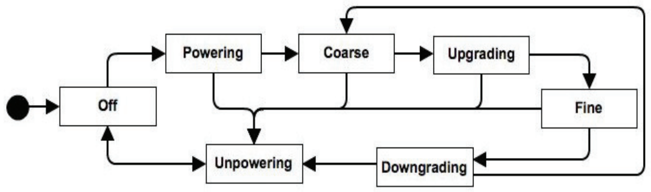


Figure 2:States and State Transitions of a Branch of GPS [1]

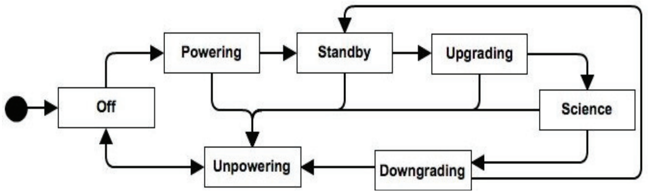


Figure 3: States and State Transitions of a Branch of PLI [1]

State transitions are very fast to accommodate time constrains for real-time satellite operations. Hence, any state transition to powering, un-powering, upgrading or downgrading takes less than one AOCS cycle. However, every state transition to off takes minimum three and maximum four AOCS cycles. Any state transition to on, coarse, fine, standby or science has a success condition if the transition gets completed during the first AOCS cycle when the condition is observed to hold. However, any state transition to on, coarse, fine, standby or science is overridden if the associated success condition is not observed to hold within a predefined number of AOCS cycles from start of the transition.

IV. CONTROLLER PHASES AND PHASE TRANSITIONS

The AOCS has two controllers -- Coarse Pointing Controller (CPC) and Fine Pointing Controller (FPC). The main objective of these two controllers is to direct the line of sight with a specified coarse accuracy and fine accuracy respectively. It is an essential requirement and must be met within given time limits. The following rules have to be observed during the controller phase transitions when a certain operational mode is reached:

1) Both controllers go to idle phase when the mode transition is set to off or standby state.

2) When the mode transition is switched to safe state, the CPC enters the running phase and the FPC remains in the idle phase.

3) When the mode transition shifts to nominal, preparation or science, the CPC goes in the idle phase and the FPC moves in the running phase.

Only one controller can be in non-idle phase at any point of time. When a controller phase has to switch from idle to running, first of all it is set to preparing. After predefined number of AOCS cycles, the controller is set to ready phase. Finally, the phase of controller is shifted to running as indicated in Figure 4. It can also be noticed that the controller can directly move to the idle phase from any of the other three phases (preparing, ready and running).

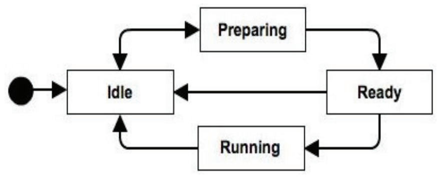


Figure 4: Phases and Phase Transitions of a Controller [1]

V. MODE TRANSITIONS

The following rules are imposed on mode transitions in order to ensure correct satellite function in nominal (fault-free) and faulty conditions:

1) When a mode transition to off or standby is completed, it is ensured that every branch in every unit is put in the off state.

2) On reaching to the safe mode, the selected branches of ES, RW and SS are set in the on state and all other branches pertaining to different units go to the off state.

3) In case of a transition to the nominal mode, the selected branch of GPS is turned in the coarse state, the concerned branches of RW, STR and THR are set to on state, and remaining every branch in every unit is put in the off state.

4) Completion of a mode transition to preparation ensures that the relevant branch of GPS is in the fine state, the chosen branch of PLI is in the standby state, the selected parts of RW, STR and THR are in the on state, and rest every branch in every unit is in the off state.

5) A mode transition to science requires that the needed branch of GPS is in the fine state, the selected branch of PLI is in the science state, the concerned branches of RW, STR and THR are in the on state, and all other branches pertaining to different units remain in the off state.

VI. FAULT TOLERANCE

Fault-tolerance should guarantee that the system continues to operate in predictable way even in case of failure of any of its components. Recovery from errors in fault-tolerant systems can be characterized as either roll forward or roll back. Forward error recovery aims at bringing the system to a new error-free state. Backward error recovery rolls back the system to some previous state before an error occurrence. In mode-rich systems, the backward error recovery is achieved via backward mode transition, i.e., mode downgrading. The mode down-gradation depends on various errors, which are explained below:

A. Branch State Transition Errors

A branch state transition error means that when some unit transitions to on state, the mode coarse, fine, standby or science gets overridden due to timeout condition. Because operation and state transition delays have to be avoided, we should time each mode transition. If a step of transition is not completed within a specified time limit, timeout signal is generated to get into a safe condition. The important error checks concerning to the branch state transitions are:

1) A branch state transition error on the redundant branch of ES, RW or SS causes a mode transition to off.

2) A mode transition to safe takes place when there is a branch state transition error on the redundant branch of GPS, STR or THR and there is no branch state transition error on the redundant branches of ES, RW and SS.

3) When a branch state transition error on the redundant branch of PLI occurs, it results into a mode transition to nominal provided that there is no branch state transition error on the redundant branches of ES, SS, GPS, RW, STR and THR.

B. Phase Transition Errors

A phase transition error or an attitude error may arise during the computations done by the selected controller. An attitude error is generated when there is a problem in the execution of an AOCS algorithm. It means that an error occurs only when one of the two controllers (i.e. CPC and FPC) is in the running phase. The key factors relating to the attitude errors are:

1) If the current mode is safe, then a non-ignored attitude error causes a transition to the off mode.

2) In case the existing mode is nominal and a non-ignored attitude error occurs, a mode transition to safe takes place.

3) A mode transition to nominal takes place when the current mode is preparation and a non-ignored attitude error is generated.

4) The generation of a non-ignored attitude error moves the mode transition to preparation with the condition that the existing mode is science.

C. Unit Reconfiguration

Each logical unit consists of two hardware units known as nominal and redundant. Initially, the nominal unit works in the active role and provides all the necessary support for normal operation of the system. The redundant unit serves as a backup resource. When an error is detected in the nominal unit, it becomes “reconfigured”. It means that the nominal unit is switched off and the redundant unit takes over the operational tasks.

The important errors that take place during the unit reconfiguration are:

1) A branch state transition error on the nominal branch of ES, SS or RW causes a reconfiguration of the unit if there is no branch state transition error on the redundant branches of ES, SS and RW.

2) A branch state transition error on the nominal branch of GPS, STR, THR or PLI causes a reconfiguration of the unit if there is no branch state transition error on the redundant branches of ES, SS, GPS, RW, STR and THR.

Figure 6 shows detailed flow chart of the implemented system.

VII. VERIFICATION

We have implemented mode-transition algorithm in SystemC language. The SystemC Verification Standard provides API for transaction based verification, constrained and weighted randomization, exception handling, and other verification tasks [4,5]. SystemC supports the use of special data types which are often used by the hardware engineers. It comes with a strong simulation kernel to enable the designers to write good test benches for easy and speedy simulation. It is extremely important because the functional verification at the system level saves a lot of money and time.

The system architecture that is implemented in SystemC is verified in the SPIN model checker. SPIN [6,7,8] is often used to verify behavior of distributed and parallel systems. PROMELA (PROcess MEta LAnguage) is a high level language which is widely used to specify systems descriptions and is fully supported by SPIN for the purpose of verification of software-based applications. SPIN PROMELA is used to carry out detailed testing and verification of design and architecture of various systems.

The simplified system architecture for AOCS is shown in Figure 5.

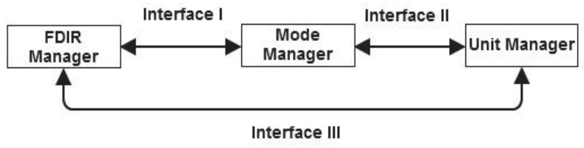


Figure 5: System Architecture [1]

An example of an interfaces between the FDIR Manager, Mode Manager and Unit Manager shown in Figure 5 are given below.

When failure occurs in the system, FDIR detects the error and issues the requests of mode transition, and then Mode Manager is responsible for mode transitions to the downgraded mode on the basis of error type. The following part of the code represents the Interface I scenario for Science Mode.

```
if (Mode==F) // Mode F: Science Mode
{       if (ES==off && SS==off && GPS==fine && STR==on &&
        RW==on && THR==on && PLI==science && CPC==idle
        && FPC==run)
        {/* The associated code describes that the conditions are valid
        for Science Mode. The current mode is Science. */}
        else if ((ES!=off || SS!=off || RW!=on) && STR==on &&
        GPS==fine && THR==on && PLI==science && CPC==idle
        && FPC==run)
        {/* The associated code describes that the conditions are not
        valid for Science Mode as error occurs on the unit branch of ES,
        SS or RW. It causes the mode transition to Off Mode. */}
        else if ((GPS!=fine || STR!=on || THR!=on) && ES==off &&
        SS==off && RW==on && PLI==science && CPC==idle &&
        FPC==run)
        {/* The associated code describes that the conditions are not
        valid for Science Mode as error occurs on the unit branch of
        GPS, STR or THR. It causes the mode transition to Safe Mode.
        */}
        else if (ES==off && SS==off && GPS==fine && STR==on
        && RW==on && THR==on && PLI!=science && CPC==idle
        && FPC==run)
        {/* The associated code describes that the conditions are not
        valid for Science Mode as error occurs on the unit branch of
        PLI. It causes the mode transition to Nominal Mode. */}
        else if (ES==off && SS==off && GPS==fine && STR==on
        && RW==on && THR==on && PLI==science &&
        (CPC!=idle || FPC!=run))
        {/* The associated code describes that the conditions are not
        valid for Science Mode as error occurs in the phase of Coarse or
        Fine Pointing Controller. It causes the mode transition to
        Preparation Mode. */}
        else
        {/* The associated code describes that no transitions take place.
        */ } }
    else
     {/* The associated code describes that it is an invalid mode. Program is
terminated.*/}
```

The SPIN's verification model successfully checks all the global mode transitions and the fault-tolerance of the system architecture. We have successfully verified forward and backward mode transitions and ensured correctness of global mode transitions with respect to component states.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed an approach to designing fault tolerant mode-rich control systems. Our work aimed at demonstrating how to design satellite control system in SystemC and verify correctness using model checking. Our approach has been demonstrated by the design of Attitude and Orbit Control System – a generic subsystem of spacecrafts.

The proposed system has been implemented in SystemC language as it is being used as a defacto verification standard in embedded systems. SystemC specification was easily aligned with Promela which works as the input language to SPIN for model checking and verification.

We have presented the design of the system and verification steps pertaining to unit branch transition errors, controller phase transition errors and unit reconfiguration.

Our work complements research done on formal modeling of mode-rich satellite systems. The formal modeling undertaken in [9,10] aimed at enabling proof-based verification of mode-rich systems modeled in Event-B. In [11] the authors perform failure modes and effect analysis of each particular mode transition to systematically design mode transition scheme. Our work aims at building a gap between formal specification and code. This motivated our choice of SystemC as a design language and model-checking based verification.

As a future work, we are planning to investigate design and verification of decentralized mode-rich systems. In particular, we will study how to ensure correctness of mode transitions as a result of negotiation between several mode managers.

REFERENCES

[1] "DEPLOY Work Package 3 - Attitude and Orbit Control System Software Requirements Document", Space Systems Finland, Ltd., December 2010.

[2] M. Heimdahl and N. Leveson, "Completeness and Consistency in Hierarchical State-Based Requirements", IEEE Transactions on Software Engineering, Vol.22, No. 6, June 1996, pp. 363-377.

[3] N. Leveson, L. D. Pinnel, S. D. Sandys, S. Koga, and J. D. Reese, "Analyzing Software Specifications for Mode Confusion Potential", Proceedings of Workshop on Human Error and System Development, C.W. Johnson, Editor, March 1997, Glasgow, Scotland, pp. 132-146.

[4] C. Ip and S. Swan, "A tutorial introduction on the new SystemC verification standard", Technical report, www.systemc.org, 2003.

[5] L. Singh and L. Drucker, "Advanced Verification Techniques : A SystemC Based Approach for Successful Tapeout", Springer, 2004.

[6] J. Katoen, "Concepts, Algorithms and Tools for Model Checking", Lecture Notes, Chapter 1: System Validation, 1999.

[7] N. A. S. A. Larc, "What is Formal Methods?", NASA Langley Methods, http://shemesh.larc.nasa.gov/fm/fm-what.html, formal methods program, 2001.

[8] Kashif Javed, Asifa Kashif, and Elena Troubitsyna, "Implementation of SPIN Model Checker for Formal Verification of Distance Vector Routing Protocol", International Journal of Computer Science and Information Security (IJCSIS), Vol 8, No 3, June 2010, USA, ISSN 1947-5500, pp. 1-6.

[9] Alexei Iliasov, Elena Troubitsyna, Linas Laibinis, Alexander Romanovsky, Kimmo Varpaaniemi, Dubravka Ilic, and Timo Latvala. Developing Mode-Rich Satellite Software by Refinement in Event B . In Proceedings of FMICS 2010, the 15th International Workshop on Formal Methods for Industrial Critical Systems, September 2010, LNCS 6371. Springer.

[10] Alexei Iliasov, Elena Troubitsyna, Linas Laibinis, Alexander Romanovsky, and Kimmo Varpaaniemi, Pauli Väisänen. Verifying Mode Consistency for On-Board Satellite Software, 2010, LNCS 6351, Computer Safety, Reliability, and Security, Pages 126-141, Springer.

[11] Yuliya Prokhorova, Elena Troubitsyna, Linas Laibinis, Kimmo Varpaaniemi, and Timo Latvala. Derivation and Formal Verification of a Mode Logic for Layered Control Systems. Asia-Pacific Software Engineering Conference. IEEE Computer, December 2011.

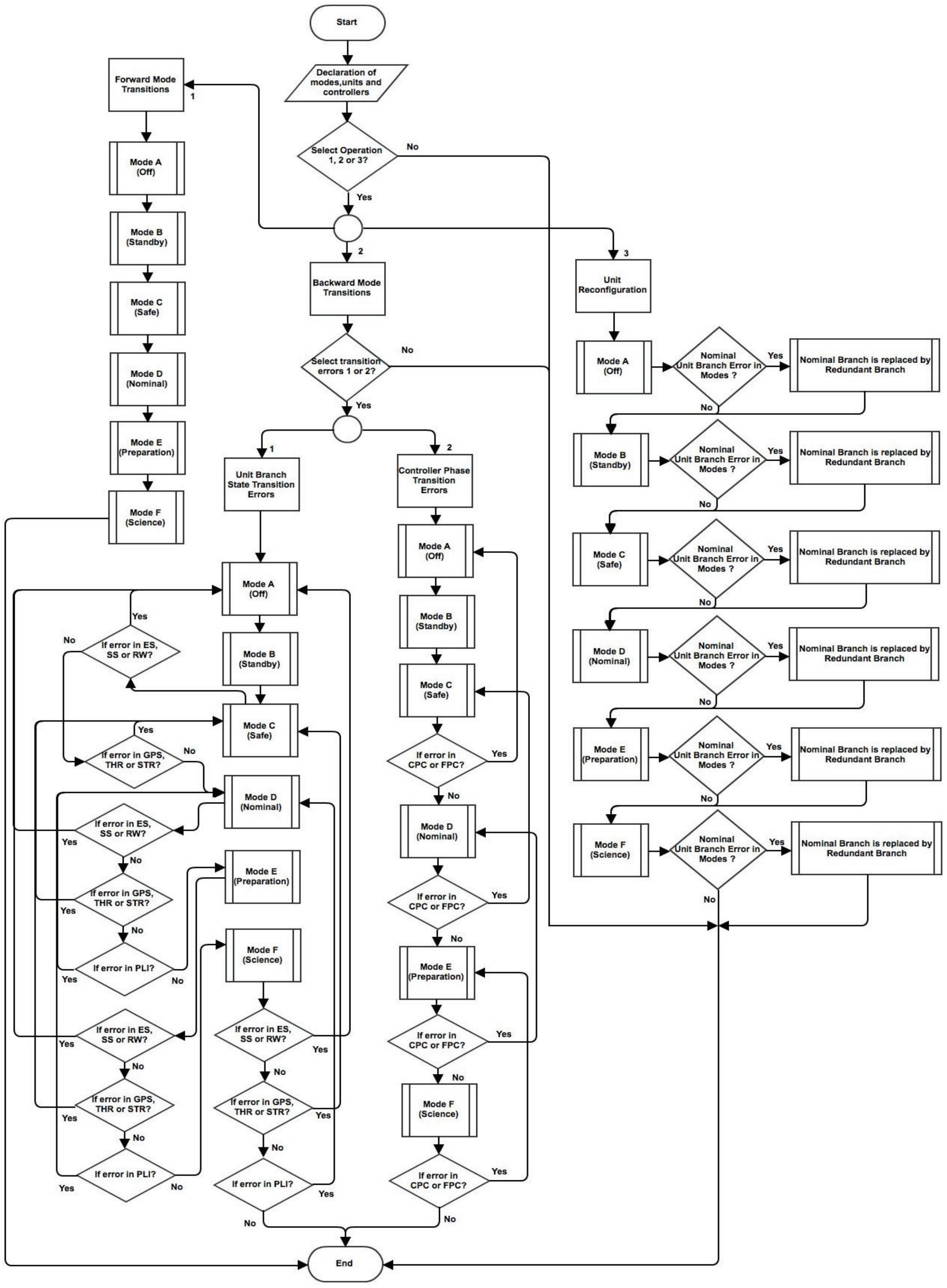

Start
Declaration of modes,units and controllers
Select Operation 1, 2 or 3?
No
Yes
1
2
3
Forward Mode Transitions
Mode A (Off)
Mode B (Standby)
Mode C (Safe)
Mode D (Nominal)
Mode E (Preparation)
Mode F (Science)
Backward Mode Transitions
Select transition errors 1 or 2?
Unit Branch State Transition Errors
Controller Phase Transition Errors
If error in ES, SS or RW?
If error in GPS, THR or STR?
If error in PLI?
If error in CPC or FPC?
Unit Reconfiguration
Nominal Unit Branch Error in Modes ?
Nominal Branch is replaced by Redundant Branch
End

Figure 6: System Flow Chart

Paper III.

K. Javed and E. Troubitsyna, “Modelling a Fault-Tolerant Distributed Satellite System”, COLLA 2012, The International Conference Advanced Collaborative Networks, Systems and Applications, pp. 35-41, June 2012, Venice, Italy.

Modelling a Fault-Tolerant Distributed Satellite System

Kashif Javed
Turku Centre for Computer Science (TUCS)
Department of Information Technologies
Abo Akademi University
Turku, FIN-20520, Finland
Kashif.Javed@abo.fi

Elena Troubitsyna
Department of Information Technologies
Abo Akademi University
Turku, FIN-20520, Finland
Elena.Troubitsyna@abo.fi

Abstract— **Ensuring correctness of a complex distributed and mode-rich collaborative satellite system is a challenging task that requires formal modeling and verification. In this paper, we propose a model of a distributed Attitude and Orbit Control System. Mode transitions in such systems are governed by a sophisticated synchronization procedure. We demonstrate how to model and verify such a procedure in order to ensure mode consistency.**

Keywords-distributed mode-rich systems; satellite software; fault tolerance; synchronization

I. INTRODUCTION

Behavior of satellite systems is often structured in terms of modes. Modes – mutually exclusive sets of system behavior define different functional profiles of the system [4,5]. An important problem associated with designing mode-rich satellite systems is to ensure correctness of mode transitions.

In this paper, we propose an approach to modeling and verification of distributed Attitude and Orbit Control System – D-AOCS [1,2]. D-AOCS is a typical example of a mode-rich collaborative system. It consists of two independent mode managers that should negotiate and coordinate their actions. Collaboration between mode managers is not trivial – faults of components might prevent the mode managers from following the agreed course of actions. As a result new negotiations would be initialized to achieve synchronization under the new conditions.

The proper synchronization is paramount for ensuring mode consistency. In general mode consistency can be seen as a high-level guarantee of a proper functioning of a distributed system deployed on the space craft. The complex collaboration procedure precedes each mode transition step.

We demonstrate how to model and verify handshaking protocol ensuring that modes are changed consistently. An important part of our modeling is fault tolerance. We demonstrate how to ensure consistency of not only nominal but also backward mode transitions, i.e., transitions to the degraded modes that are responsible for error recovery. The novelty of the proposed approach is in treating fault tolerance of collaborative systems as a problem of ensuring mode consistency.

Section II explains the state-of-the-art of AOCS structure. Section III presents AOCS architecture covering unit manager, mode manager, and fault tolerance. Handshake protocol is explained in detail in Section IV and the proposed system design using handshake is discussed in Section V. Finally, Section VI provides a brief summary of conclusions and future work.

II. STATE-OF-THE-ART STRUCURE

Attitude and Orbit Control System (AOCS) is extensively used in the design and development of modern satellites. The major objective of an AOCS is to ensure controlled movements of the satellites in order to maintain required attitude and remain in the given orbit. As disturbance of the atmosphere tends to change orientation of the satellites, there is a serious need to continuously control and monitor its attitude. A number of sensors are employed to collect data for the purpose of controlling attitude. Appropriate corrective measures are taken by the actuators to keep the right path and orbit whenever there is change detected in the data sent by the sensors. This requirement is very essential for supporting needs of payload instruments as well as for the fulfillment of satellite's mission.

The top level schema of an AOCS is shown in Figure 1.

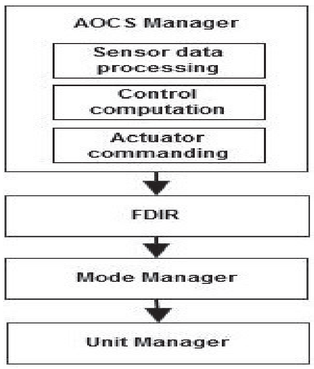


Figure 1: Top Level Schema of AOCS

The AOCS manager consists of three components (i.e., sensor data processing, control computation and actuator commanding). Control computation part handles all the data and measurements using state-of-the-art control algorithms and gives commands to the actuators for ensuring correct path and attitude. Different types of controllers are required for completion of specific mission stages. Normally, two control algorithms are used during the operational mode of the satellites.

Each unit of the satellite has a unique status (i.e., free, reserved, or locked) for its usage while avoiding conflicts during reconfiguration [10]. An actuator, payload or sensor remains free when it is idle in any mode. The reserved status means that a sensor/actuator/payload is to be used shortly but it is not yet ready. When any unit is allocated and is being used for its required operation, then it is turned into the locked status.

III. ARCHITECTURE

In this paper, we consider a distributed version of Attitude and Orbit Control System. Attitude and Orbit Control System (AOCS) [1] is a generic component of a spacecraft. Behavior of AOCS is structured using the notion of modes – mutually exclusive sets of system behavior. The complexity of designing distributed AOCS lies in the fact that mode management is decentralized, i.e., it is performed by several mode managers. Distributed AOCS (D-AOCS) has a complex architecture. It consists of AOCS Manager, Unit Manager, Several Mode Managers and FDIR (Failure Detection, Isolation and Recovery) Manager. AOCS Manger deals with two controllers -- Control Pointing Controller (CPC) and Fine Pointing Controller (FPC). The purposes of CPC and FPC are to direct line of sight as well as to provide coarse and fine accuracy. Unit level state transitions and mode transitions are managed by Unit Manager and Mode Manager respectively. FDIR Manager ensures handling of branch state transition errors and controller phase transition errors [2]. Two managers -- Mode Manager 1 (MM1) and Mode Manager 2 (MM2) are responsible for the global mode logic of D-AOCS. The architecture of Unit Manager and Mode Managers is described below.

A. *Unit Manager*

The Unit Manager in D-AOCS organizes the internal states of the units. The components of Unit Manager are supervised by the Mode Manager. The controlled units include Earth Sensor (ES), Sun Sensor (SS), Star Tracker (STR), Global Positioning System (GPS), Reaction Wheel (RW), Thruster (THR) and Payload Instrument (PLI). All unit components are responsible for mode synchronization, decision making on unit states, performing branch state transitions and unit reconfiguration [4,5]. SS, STR, GPS, RW and PLI provide data to the AOC Manager. RW and THR execute the commands from AOC Manager. These units are also responsible for detection and reporting the branch state transition errors [1].

Every unit consists of two identical branches -- the nominal and redundant ones. At any instance of time only one branch is active. A unit branch in the 'on' state is always assigned locked status and the unit branch in 'off' state has unlocked status. There are six states of unit components -- on, off, coarse, fine, standby and science. The internal states of ES, SS, STR, RW and THR are either 'on' or 'off'. Three possible GPS's operational states are 'off', 'coarse' and 'fine'. PLI's state can be in 'off', 'standby' or 'science' [3].

B. *Mode Managers*

The global mode transitions are managed by the two mode managers -- MM1 and MM2. Each mode manager's controls different units. Each mode manager is responsible for checking the preconditions of mode transitions, managing the controllers and the units, and initiating and completing the mode transitions. The global modes are correspondingly Off, Standby, Safe, Nominal, Preparation, and Science [10]. Below we give a brief description of each mode:

Off: After the central data management unit completes booting of AOCS software, the satellite instantly goes into the off mode.

Standby: The process of separation of the satellite from the launcher is monitored during the standby mode.

Safe: After successful separation from the launcher, the satellite switches to the safe mode. The satellite obtains a stable attitude and the CPC is activated.

Nominal: After transition to this mode, FPC is activated, while CPC is switched off. PLI is actoviated to provide measurements for FPC.

Preparation: FPC is achieved in the preparation mode and PLI gets ready to perform the necessary tasks.

Science: PLI carries out the required tasks and stays in science mode till the desired tasks are completed.

MM1 and MM2 communicate with each other to synchronize on mode transitions that are performed in parallel. Let us describe the scenario of mode transitions. After a mode transition to off or standby is done, every unit branch goes to off state and both controllers are idle. After that, both mode managers communicate with each other. If there is no error then transition to the next mode is executed. When the mode is switched to safe, the selected branches of ES, SS and RW are turned to 'on' state and only FPC remains idle. Both mode managers send messages to inform each other that no error occured in the given modes. After a handshake, they perform the mode transition to the nominal mode. In a mode transition to the nominal mode, the required branches of RW, STR and THR are put to the 'on' state and GPS is put into the'coarse' state. The messages sent and received by the mode managers notify each mode manager that no unit or controller error has occured. Then the preparation mode is reached, the concerned branches of RW, STR & THR are in the 'on' state and GPS & PLI are in the 'fine' state and 'standby' state respectively. They ensure the correctness of the modes in MM1 and MM2 and make a transition to the science mode. In case of the science mode, the preffered branch of PLI operates with 'science' state. All other units keep their previous state. When a mode transition goes to nominal, preparation or science mode, only CPC remains idle. MM1 and MM2 both inform each other regarding success of mode transition.

C. Fault Tolerance

Fault tolerance aims at providing the system with the means to continue its function in spite of errors of its components. In the D-AOCS backward error recovery is adopted, i.e., if an error occurs, the system gets back to some previous state to handle the error. The roll back error recovery is implemented by the backward mode transitions. The mode roll-back depends on branch state transition errors and phase transition errors.

There are different aspects relating to the branch state transition errors. When a branch state transition error on the redundant branch of ES, RW or SS occurs and there is no error in the remaining redundant branches, then the mode goes back to off mode. If the redundant branch of GPS, STR or THR gets corrupted, it results a mode transition to safe. A mode transition to nominal takes place when there is a branch state transition error on the redundant branch of PLI.

The important error checks are incorporated to deal with the attitude or phase transitions. When the current mode is safe and a non-negligible phase error is produced, it results in a mode transition to off. If the phase error is generated in the nominal, then it goes back to safe. In case the existing mode is preparation and a phase error occurs, a mode transition to nominal takes place. A mode transition to preparation takes place when a phase error occurs in the science mode [3].

In case of unit reconfiguration, a branch state transition error on the nominal branch of any unit causes a unit reconfiguration if there is no branch state transition error on the redundant branches of that particular unit.

If the mode task is not completed within a given time interval or multiple errors occur in the unit branches and controller phases, then timeout signal is produced for safe condition.

IV. HANDSHAKE PROTOCOL

Handshaking is a process in which connection is established among two processes and information is transferred from one process to another without the need for human involvement to set constraints. MM1 and MM2 do handshake with each other to update the condition of their modes. Different scenarios of handshake protocol are explained covering the following key points:

If all conditions of unit states and controller phases within each mode of MM1 and MM2 fulfill their requirements, then mode managers pass the ‘no error’ message to notify that the mode is in the error-free state. It results in the forward mode transition, i.e., the mode manager switches the current mode to the next mode as described in Section III.

If an error occurs during a mode transition of MM1 and there is no error in the mode of MM2, then MM1 sends an ‘error’ message to MM2. MM1 executes error recovery, i.e., starts backward mode transtion according to the Section III. Until the error recovery of MM1 is not completed, MM2 keeps on waiting. After the successful error recovery, both mode managers proceed to the next mode.

When an error occurs only in the mode of MM2, then MM1 receives an ‘error’ message from MM2. MM1 waits until error has been recovered in MM2. The mode managers switch to next mode after receiving the information from MM2 that the error is recoverd.

Upon receiving an ‘error’ message from MM1 and MM2 simultaneously, error recovery starts in both mode managers as mentioned in Section III. The backward mode transitions are executed in MM1 and MM2. After achieving the successful recovery, mode managers move to the next mode.

There are two types of errors -- the unit branch state transition errors and controller phase transition errors. Handshaking algorithm for handling such type of errors is quite complex as specified below:

```
void handshake(int u_MM1, int u_MM2,int c_MM1,int c_MM2) {
// ‘u’ denotes unit error flag and ‘c’ denotes controller error flag
 if (u_MM1==0&&u_MM2==0&&c_MM1==0&&c_MM2==0) {
     /* The associated code illustrates that no error occurs in the unit
     branchs of ES, SS, RW, GPS, STR, THR or PLI and controller
     phase of CPC or FPC in the given mode of MM1 and MM2. It
     accounts the forward mode transition according to the Section
     III. */}
elseif(u_MM1==1&&u_MM2==0&&c_MM1==0&&c_MM2==0) {
     /* The associated code illustrates that an error occurs in the unit
     branch of ES, SS, RW, GPS, STR, THR or PLI in the given
     mode of MM1. It accounts the backward mode transition
     according to the Section III. MM2 stays on waiting until an
     error is recovered. */}
elseif(u_MM1==0&&u_MM2==1 &&c_MM1==0&&c_MM2==0) {
     /* The associated code illustrates that an error occurs in the unit
     branch of ES, SS, RW, GPS, STR, THR or PLI in the given
     mode of MM2. It accounts the backward mode transition
     according to the Section III. MM1 stays on waiting until an
     error is recovered. */}
elseif(u_MM1==1&&u_MM2==1&&c_MM1==0&&c_MM2==0) {
     /* The associated code illustrates that an error occurs in the unit
     branch of ES, SS, RW, GPS, STR, THR or PLI in the given
     mode of both mode managers. MM1 and MM2 account the
     backward mode transition according to the Section III. */}
elseif(u_MM1==0&&u_MM2==0&&c_MM1==1&&c_MM2==0) {
     /* The associated code illustrates that an error occurs in the
     controller phase of CPC or FPC in the given mode of MM1. It
     accounts the backward mode transition according to the Section
     III. MM2 stays on waiting until an error is recovered. */}
elseif(u_MM1==0&&u_MM2==0&&c_MM1==0&&c_MM2==1) {
     /* The associated code illustrates that an error occurs in the
     controller phase of CPC or FPC in the given mode of MM2. It
     accounts the backward mode transition according to the Section
     III. MM1 stays on waiting until an error is recovered. */}
elseif(u_MM1==0&&u_MM2==0&&c_MM1==1&&c_MM2==1) {
     /* The associated code illustrates that an error occurs in the
     controller phase of CPC or FPC in the given mode of both mode
```

```
        managers. MM1 and MM2 account the backward mode
        transition according to the Section III. */}
else {
        /* The associated code describes that it is an invalid condition.
        Program is terminated.*/}              }
```

V. PROPOSED SYSTEM DESIGN USING HANDSHAKE

The proposed system design has been implemented using SystemC. SystemC can be used at system level for functional verification. The framework also supports event driven simulation environments [6]. It offers application program interface for transaction based verification, handling exceptions and verification tasks [7]. The system model consists of six defined modes named as A (Off), B (Standby), C (Safe), D (Nominal), E (Preparation) and F (Science). Three different operations have been implemented (i.e., forward mode transitions, backward mode transitions, and unit reconfiguration). The flow chart given in Figure 3 describes detailed design structure for only one transition from Mode E to Mode F of the system. When the system reaches to Mode E, it checks the error in the Mode E of both mode managers. Figure 3 shows the operations regarding error condition according to the scenarios and backward mode transitions according to the error types (Unit branch error (redundant/nominal) and controller phase error) they are discussed in Section IV and Section III respectively.

After necessary declarations of modes, units and controllers, the verification of the system are described in the following sections.

A. Verification of Forward Mode Transition

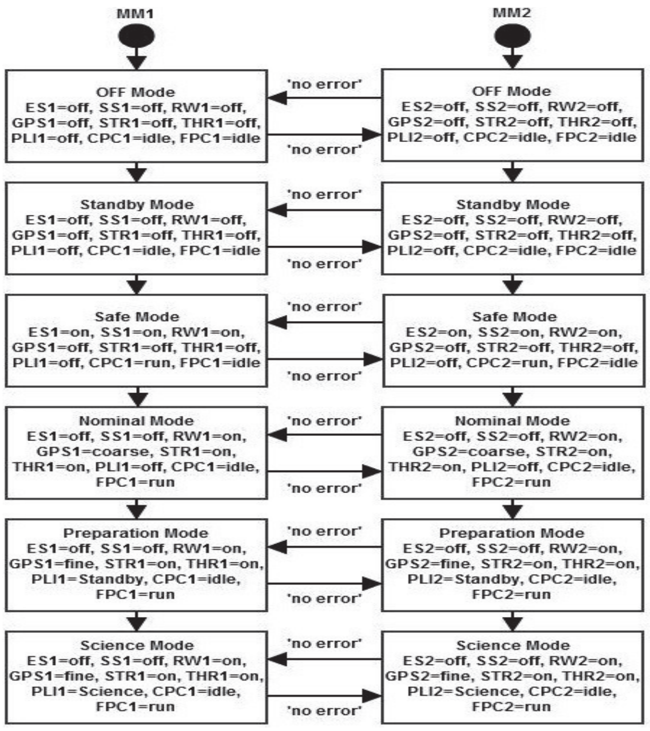

MM1
MM2
OFF Mode
ES1=off, SS1=off, RW1=off,
GPS1=off, STR1=off, THR1=off,
PLI1=off, CPC1=idle, FPC1=idle
'no error'
'no error'
OFF Mode
ES2=off, SS2=off, RW2=off,
GPS2=off, STR2=off, THR2=off,
PLI2=off, CPC2=idle, FPC2=idle
Standby Mode
ES1=off, SS1=off, RW1=off,
GPS1=off, STR1=off, THR1=off,
PLI1=off, CPC1=idle, FPC1=idle
'no error'
'no error'
Standby Mode
ES2=off, SS2=off, RW2=off,
GPS2=off, STR2=off, THR2=off,
PLI2=off, CPC2=idle, FPC2=idle
Safe Mode
ES1=on, SS1=on, RW1=on,
GPS1=off, STR1=off, THR1=off,
PLI1=off, CPC1=run, FPC1=idle
'no error'
'no error'
Safe Mode
ES2=on, SS2=on, RW2=on,
GPS2=off, STR2=off, THR2=off,
PLI2=off, CPC2=run, FPC2=idle
Nominal Mode
ES1=off, SS1=off, RW1=on,
GPS1=coarse, STR1=on,
THR1=on, PLI1=off, CPC1=idle,
FPC1=run
'no error'
'no error'
Nominal Mode
ES2=off, SS2=off, RW2=on,
GPS2=coarse, STR2=on,
THR2=on, PLI2=off, CPC2=idle,
FPC2=run
Preparation Mode
ES1=off, SS1=off, RW1=on,
GPS1=fine, STR1=on, THR1=on,
PLI1=Standby, CPC1=idle,
FPC1=run
'no error'
'no error'
Preparation Mode
ES2=off, SS2=off, RW2=on,
GPS2=fine, STR2=on, THR2=on,
PLI2=Standby, CPC2=idle,
FPC2=run
Science Mode
ES1=off, SS1=off, RW1=on,
GPS1=fine, STR1=on, THR1=on,
PLI1=Science, CPC1=idle,
FPC1=run
'no error'
'no error'
Science Mode
ES2=off, SS2=off, RW2=on,
GPS2=fine, STR2=on, THR2=on,
PLI2=Science, CPC2=idle,
FPC2=run

Figure 2: Forward Mode Transitions

When all the units are in off state, controller phases are in the idle phase, and no unit reconfiguration is in progress, then current mode is A in MM1 and MM2. The unit/controller error flag is set to low and mode managers exchange the information ('no error' message) of error-free mode status. After this, the mode moves forward to the next mode (i.e., Mode B) in MM1 and MM2. Hence, when all conditions of unit states and controller phases within each mode of each manager fulfill their requirements, mode managers update each other about the error-free mode conditions. Then the current mode switches to the next mode within each mode manager until it completes its operation after Mode F. Figure 2 illustrates the implemented procedure that corresponds to the forward mode transition for MM1 and MM2.

B. Verification of the Steps in the Backward Mode Transition

The backward mode transition depends on the two types of errors (i.e., unit branch state transition error and controller phase transition error). Handshaking procedure for handling these errors is given below.

1) Verification of the Steps in Unit Branch State Transition Error

Following part of the code segment describes the unit branch transition error in case of Mode E as shown in Figure 2. If there is an error in ES, SS or RW of MM1, MM1 switches to Mode A. If an error occurs in GPS, THR or STR of MM2, MM2 return to Mode C. However, if PLI gets an error in both mode managers, MM1 and MM2 both go back to Mode D. Before backward transition to the desired mode, the messages exchange information between the effected mode manager and the error-free mode manager to acknowledge the error status.

```
// Variable declarations
int FPC1,CPC1,FPC2,CPC2,u_MM1,c_MM1,u_MM2,c_MM2;
// unit states
const int off=0;const int on=1;const int coarse=2;
const int fine=3;const int unit=0;const int Standby=4;
const int Science=5;const int idle=0;const int run=1;
const int A=1;const int B=2;const int C=3;
const int D=4;const int E=5;const int F=6;
/* Each unit has two branches i.e., Nominal and Redundant,
here we deal with redundant branch of the units. */
int ES1,SS1,GPS1,STR1,RW1,THR1,PLI1; // MM1 Units
int ES2,SS2,GPS2,STR2,RW2,THR2,PLI2; // MM2 Units
        if(mode==E) {// Preparation Mode
        if((ES1!=off || SS1!=off || RW1!=on) && STR1==on &&
        GPS1==fine && THR1==on && PLI1== standby &&
        CPC1==idle && FPC1==run && ES2==off &&
        SS2==off && RW2==on && STR2==on &&
        GPS2==fine && THR2==on && PLI2== standby &&
        CPC2==idle && FPC2==run){
                u_MM1=1;c_MM1=0;
                u_MM2=0;c_MM2=0;
                /* The remaining part of the code, by calling the
                handshake protocol function on the basis of unit and
                controller error flag, is mentioned in Section IV.*/}
        else if(ES1==off && SS1==off && RW1==on &&
        STR1==on && GPS1==fine && THR1==on &&
PLI1==standby && CPC1==idle && FPC1==run &&
ES2==off && SS2==off && RW2==on && (STR2!=on ||
GPS2!=fine || THR2!=on) && PLI2==standby &&
CPC2==idle && FPC2==run){
    u_MM1=0;c_MM1=0;
    u_MM2=1;c_MM2=0;
    /* The remaining part of the code, by calling the
    handshake protocol function on the basis of unit and
    controller error flag, is mentioned in Section IV.*/}
else if(ES1==off && SS1==off && RW1==on &&
STR1==on && GPS1==fine && THR1==on &&
PLI1!=standby && CPC1==idle && FPC1==run &&
ES2==off && SS2==off && RW2==on && STR2==on
&& GPS2==fine && THR2==on && PLI2!=standby &&
CPC2==idle && FPC2==run){
    u_MM1=1;c_MM1=0;
    u_MM2=1;c_MM2=0;
    /* The remaining part of the code, by calling the
    handshake protocol function on the basis of unit and
    controller error flag, is mentioned in Section IV.*/}
else{
    /* The associated code describes that no transitions
    take place. */ }            }
else cout<<" Program is terminated.";
```

2) *Verification of the Steps in Controller Phase Transition Errors*

When CPC and FPC do not fulfill the requirement of mode of any mode manager, the error flag is set to high and the affected mode manager is downgraded to previous mode after utilizing the handshake protocol by sending message to error-free mode manager. In case the phase of controllers in the given mode of both mode managers is corrupted, then both managers do the backward mode transition at once after acknowledging each other. The following portion of the code represents the scenario of phase transition for Mode E as illustrated in Figure 2.

```
//Variables are declared in the previous section.
if(mode==E) {// Preparation Mode
if(ES1==off && SS1==off && RW1==on &&
STR1==on && GPS1==fine && THR1==on &&
PLI1==standby && CPC1!=idle && FPC1==run &&
ES2==off && SS2==off && RW2==on && STR2==on
&& GPS2==fine && THR2==on && PLI2==standby &&
CPC2==idle && FPC2==run){
    u_MM1=0;c_MM1=1;
    u_MM2=0;c_MM2=0;
    /* The remaining part of the code, by calling the
    handshake protocol function on the basis of unit and
    controller error flag, is mentioned in Section IV.*/}
else if(ES1==off && SS1==off && RW1==on &&
STR1==on && GPS1==fine && THR1==on &&
PLI1==standby && CPC1==idle && FPC1==run &&
ES2==off && SS2==off && RW2==on && STR2==on
&& GPS2==fine && THR2==on && PLI2==standby &&
CPC2==idle && FPC2!=run){
    u_MM1=0;c_MM1=0;
    u_MM2=0;c_MM2=1;
    /* The remaining part of the code, by calling the
    handshake protocol function on the basis of unit and
    controller error flag, is mentioned in Section IV.*/}
else if(ES1==off && SS1==off && RW1==on &&
STR1==on && GPS1==fine && THR1==on &&
PLI1==standby && CPC1==idle && FPC1!=run &&
ES2==off && SS2==off && RW2==on && STR2==on
&& GPS2==fine && THR2==on && PLI2==standby &&
CPC2!=idle && FPC2==run){
    u_MM1=0;c_MM1=1;
    u_MM2=0;c_MM2=1;
    /* The remaining part of the code, by calling the
    handshake protocol function on the basis of unit and
    controller error flag, is mentioned in Section IV.*/}
else{
    /* The associated code describes that no transitions
    take place. */ }            }
else cout<<" Program is terminated.";
```

C. *Verification of the Steps in Unit Reconfiguration*

If error exists on nominal unit branch at any mode of MM1 or MM2, then it is replaced by redundant unit branch in the given mode of mode manager. The unit reconfiguration is done to complete the remaining operation of the system. Unit reconfiguration is, however, a burden on the system and takes some time while switching from nominal branch to redundant branch of the unit. In case of the nominal unit branch in the given mode of both mode managers is corrupted, then unit reconfiguration is done in both mode manager after exchanging the information between the mode managers regarding unit reconfiguration.

The following piece of the code shows the scenario of unit reconfiguration for Mode E as shown in Figure 2.

```
//Variables are declared in the previous section. In reconfiguration
module, we also deal with nominal branch of the units. So, both
branches of the unit are declared separately.
//Nominal branches of MM1 and MM2
int N_ES1, N_SS1, N_RW1, N_GPS1, N_STR1, N_THR1, N_PLI1;
int N_ES2, N_SS2, N_RW2, N_GPS2, N_STR2, N_THR2, N_PLI2;
//Redundant branches of MM1 and MM2
int R_ES1, R_SS1, R_RW1, R_GPS1, R_STR1, R_THR1, R_PLI1;
int R_ES2, R_SS2, R_RW2, R_GPS2, R_STR2, R_THR2, R_PLI2;
if (mode==E) { //  Preparation Mode
if((N_ES1!=off || N_SS1!=off || N_RW1!=on) && R_ES1==off &&
R_SS1==off && R_RW1==on && N_ES2==off && N_SS2==off
&& N_RW2==on && R_ES2==off && R_SS2==off &&
R_RW2==on ) {
    u_MM1=1;c_MM1=0;
    u_MM2=0;c_MM2=0;
    /* The remaining part of the code, by calling the handshake
    protocol function on the basis of unit and controller error flag, is
    mentioned in Section IV.*/}
else if(N_GPS1==fine && N_STR1==on && N_THR1==on &&
N_PLI1==standby && R_GPS1==fine && R_STR1==on &&
R_THR1==on && R_PLI1==standby && (N_GPS2!=fine ||
N_STR2!=on || N_THR2!=on || N_PLI2!=standby) &&
R_GPS2==fine && R_STR2==on && R_THR2==on &&
R_PLI2==standby) {
    u_MM1=0;c_MM1=0;
    u_MM2=1;c_MM2=0;
    /* The remaining part of the code, by calling the handshake
    protocol function on the basis of unit and controller error flag, is
    mentioned in Section IV.*/}
else if((N_ES1!=off || N_SS1!=off || N_RW1!=on) && R_ES1==off
&& R_SS1==off && R_RW1==on && (N_ES2==off || N_SS2!=off
|| N_RW2!=on) && R_ES2==off && R_SS2==off && R_RW2==on
) {   u_MM1=1;c_MM1=0;
    u_MM2=1;c_MM2=0;
```

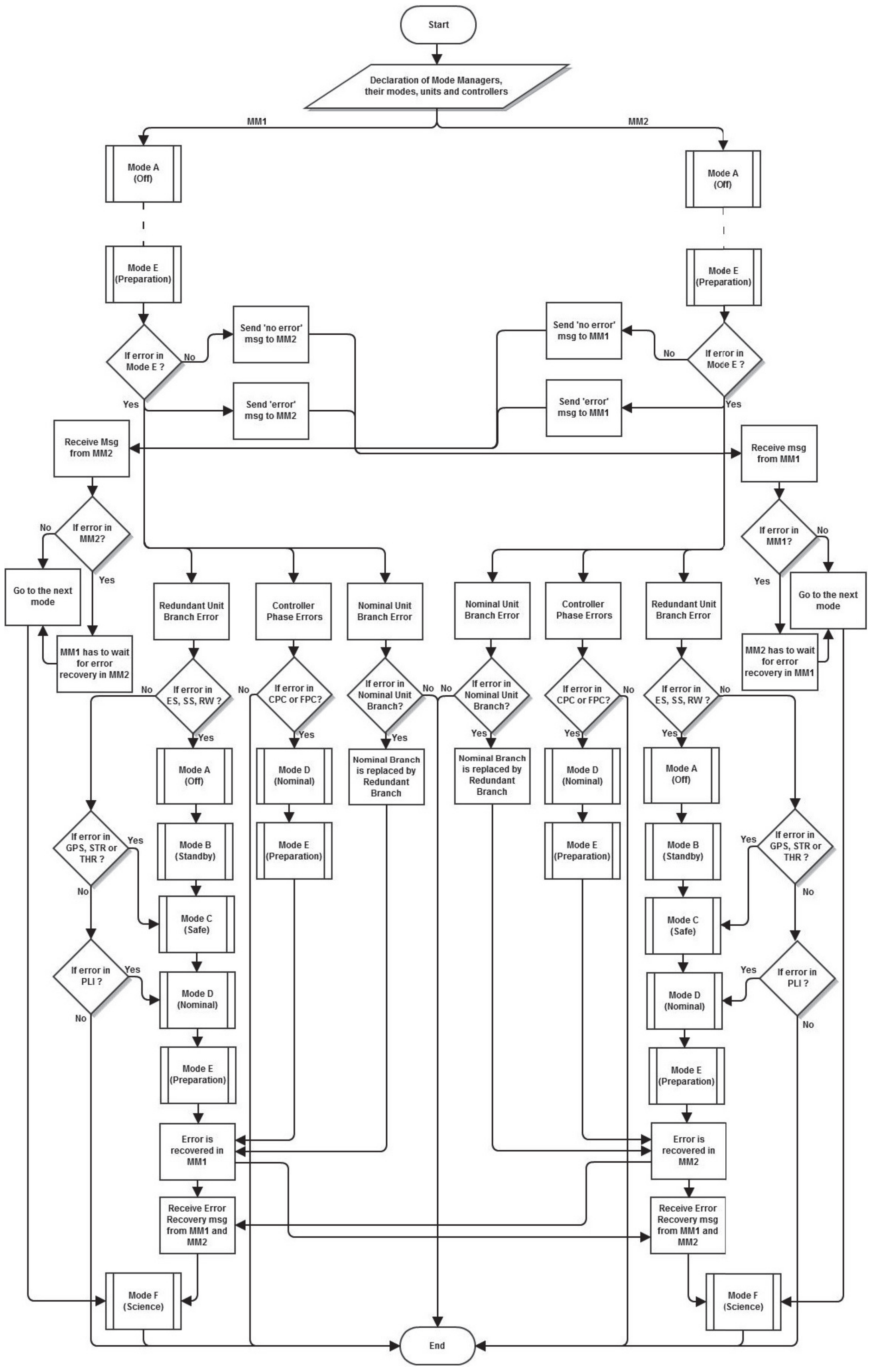

Start
Declaration of Mode Managers, their modes, units and controllers
MM1
MM2
Mode A (Off)
Mode E (Preparation)
If error in Mode E ?
No
Yes
Send 'no error' msg to MM2
Send 'error' msg to MM2
Send 'no error' msg to MM1
Send 'error' msg to MM1
Receive Msg from MM2
Receive msg from MM1
If error in MM2?
If error in MM1?
Go to the next mode
MM1 has to wait for error recovery in MM2
MM2 has to wait for error recovery in MM1
Redundant Unit Branch Error
Controller Phase Errors
Nominal Unit Branch Error
If error in ES, SS, RW ?
If error in CPC or FPC?
If error in Nominal Unit Branch?
Nominal Branch is replaced by Redundant Branch
Mode B (Standby)
Mode C (Safe)
Mode D (Nominal)
If error in GPS, STR or THR ?
If error in PLI ?
Error is recovered in MM1
Error is recovered in MM2
Receive Error Recovery msg from MM1 and MM2
Mode F (Science)
End

Figure 3: System flow chart for Mode E to Mode F

```
        /* The remaining part of the code, by calling the handshake
        protocol function on the basis of unit and controller error flag, is
        mentioned in Section IV.*/}
  else if((N_ES1!=off || N_SS1!=off || N_RW1!=on) && R_ES1==off
  && R_SS1==off && R_RW1==on && (N_ES2==off || N_SS2!=off
  || N_RW2!=on) && R_ES2==off && R_SS2==off && R_RW2==on
  ) {   u_MM1=1;c_MM1=0;
        u_MM2=1;c_MM2=0;
        /* The remaining part of the code, by calling the handshake
        protocol function on the basis of unit and controller error flag, is
        mentioned in Section IV.*/}
  else{
        /* The associated code describes that no transition takes place.
        */ }          }
else    cout<<" Program is terminated.";}
```

Our verification efforts are focused on checking correctness of mode syncornization and verification of the proposed collaboration scheme. To obtain quantitative measures of the performance of the discussed protocol we would need to further refine our specification and to integrate model of hardware platform in the loop. We are planning to perform quantitative evaluation as a part of the future work.

VI. Conclusions and Future Work

In this paper, we demonstrated how to model and verify distributed satellite systems with complex mode transition logic. Our approach is validated by a case study – design of a distributed Attitude and Orbit Control System.

The proposed system has been implemented in SystemC language. SystemC specification can be easily interfaced with various model checking techniques to perform formal verification. The work presented in this paper extends our previous work done on modeling centralized mode-rich system. In the current approach, we have put the main focus on mode synchronization aspect and demonstrated how to achieve mode consistency via handshaking protocol.

Our work complements research done on formal modeling of mode-rich satellite systems. The formal modeling proposed by Iliasov et al. [8,9] focused on proof-based verification of centralized AOCS. Formal modeling of the distributed architecture presented in our paper is a completely novel aspect.

As a future work, we are planning to investigate how to interface architectural modeling with our design approach.

References

[1] “DEPLOY Work Package 3 - Software Requirements Document for Distributed System for Attitude and Orbit Control for a Single Spacecraft”, Space Systems Finland, Ltd., June 2011[retrieved: November, 2011].

[2] “DEPLOY Work Package 3 - Attitude and Orbit Control System Software Requirements Document”, Space Systems Finland, Ltd., December 2010 [retrieved: January, 2012].

[3] J. Kashif, and E. Troubitsyna, "Designing a Fault-Tolerant Satellite System in SystemC", ICONS 2012, The Seventh International Conference on Systems, XPS Press, pp. 49-54, March 2012.

[4] M. Heimdahl, and N. Leveson, “Completeness and Consistency in Hierarchical State-Based Requirements”, IEEE Transactions on Software Engineering, Vol.22, No. 6, pp. 363-377, June 1996.

[5] N. Leveson, L. D. Pinnel, S. D. Sandys, S. Koga, and J. D. Reese, “Analyzing Software Specifications for Mode Confusion Potential”, Proceedings of Workshop on Human Error and System Development, C.W. Johnson, Editor, Glasgow, Scotland, pp. 132-146, March 1997.

[6] N. Blanc, D. Kroening, and N. Sharygina, “Scoot: A Tool for the Analysis of SystemC Models”. TACAS'08/ETAPS'08 Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the Construction and Analysis of Systems, Springer-Verlag, Berlin, Heidelberg, pp. 467–470, 2008.

[7] L. Singh, and L. Drucker, “Advanced Verification Techniques: A SystemC Based Approach for Successful Tapeout”, Kluwer Academic Publishers, Springer, 2004.

[8] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, D. Ilic, and T. Latvala, “Developing Mode-Rich Satellite Software by Refinement in Event B”. In: Proc.of FMICS 2010, the 15th International Workshop on Formal Methods for Industrial Critical Systems, Lecture Notes for Computer Science, Springer, 2010.

[9] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, P. Väisänen, D. Ilic, and T. Latvala, “Verifying Mode Consistency for On-Board Satellite Software”. In Proc. of SAFECOMP 2010, The 29th International Conference on Computer Safety, Reliability and Security, September 14-17, Vienna, Austria, Lecture Notes for Computer Science, Springer, September 2010.

[10] “DEPLOY deliverable D20 – Pilot Deployment in the Space Sector”, Space Systems Finland, Ltd., January 2010 [retrieved: March, 2012].

Paper IV.

K. Javed and E. Troubitsyna, “A Case Study in Modelling a Fault Tolerant Satellite System Implementing Dynamic Reconfiguration via Handshake”, ICSEA2012, The Seventh International Conference on Software Engineering Advances, pp. 44-49, November 2012, Lisbon, Portugal.

A Case Study in Modeling a Fault-tolerant Satellite System through Implementation of Dynamic Reconfiguration via Handshake

Kashif Javed
Turku Centre for Computer Science (TUCS)
Department of Information Technologies
Abo Akademi University
Turku, FIN-20520, Finland
Kashif.Javed@abo.fi

Elena Troubitsyna
Department of Information Technologies
Abo Akademi University
Turku, FIN-20520, Finland
Elena.Troubitsyna@abo.fi

Abstract— **Fault tolerance of satellite systems is critical for ensuring the success of the space mission. To minimize redundancy of the on-board equipment, the satellite systems should rely on dynamic reconfiguration in case of failures of some of their components. In this paper, modeling and implementation of a handshake procedure has been presented that becomes a crucial part of the dynamic reconfiguration process of a satellite subsystem for data processing. The model for handshake methodology is specialized software for quickly and successfully recovering from the crisis and failure situation of the satellite system.**

Keywords – dynamic reconfiguration; fault tolerance; advanced software for handshake procedure; modeling and verification.

I. INTRODUCTION

To ensure high reliability during long-term missions, the satellite systems rely on redundancy to achieve fault tolerance and guarantee that the system would be able to deliver its services despite component failures. However, the use of redundancy in the satellites is restricted by the constraints put on the weight and volume of the on-board equipment.

Despite a careful analysis performed to ensure the desired degree of reliability, recently one of the satellites has experienced a double-failure problem with a system that samples and packages scientific data [6]. The system consisted of two identical modules. When one of the subcomponents of the first module failed, the system switched to the use of the second module. However, after a while a subcomponent of the spare module also failed, so it became impossible to produce scientific data. In order to avoid failure of the entire mission, the company controlling the operation of the system has invented a solution that relies on healthy subcomponents of both modules and provides complex communication mechanism based on the handshake procedure to restore functioning and to resume production of scientific data.

In this paper, we present a case study in modeling and implementation of Control and Data Management Unit (CDMU) [1] - a generic subsystem of satellites. In particular, we focus on modeling fault tolerance aspect of the system that is implemented as a handshake procedure between two redundant systems. This mechanism is introduced to achieve the dynamic reconfiguration. For this purpose, a formal model of the handshake procedure has been designed and implemented in Promela. Handshake modeling is an advanced software application to deal with dynamic reconfiguration for ensuring fault-tolerance when the mission-critical satellite system encounters faults in its component and errors in data communication.

This paper is structured as follows. Section II describes the state-of-the-art model of CDMU and Section III presents the architecture of the control and data management unit. Section IV describes the handshake procedure performed to reconfigure the system from simple redundant two-module architecture to the Master-Slave architecture. The proposed system model for handshake is explained in Section V covering all relevant details of master and slave modules. Section VI discusses the handshake model between the two reconfiguration modules that has been implemented and verified using SPIN/PROMELA. Finally, conclusions and future work are summarized in Section VII.

II. STATE-OF-THE-ART MODEL

CDMU is a state-of-the-art platform to monitor and control the satellites system and to organize the collected on-board data. The major objective of CDMU is to acquire and transmit the data to the ground after carrying out appropriate processing. Moreover, it also distributes and decodes the given commands to its all redundant systems consisting of processor, reconfiguration and telemetry modules. Whenever any failure or data error takes place during the operation of the satellite system, there is an emergent requirement to dynamically reconfigure the components of CDMU for its smooth and crisis-free control and data management. Processing and storing of satellite data at the right time is of top-most importance during the working and recovery procedure of the proposed system. In case of experiencing any failure, the implemented CDMU structure and the developed model of handshake procedure immediately adapts to the well-defined and specialized switchover mechanism for shifting from one redundant processor to another in order to reconfigure and provide safe operation of the satellite system during its critical mission.

III. ARCHITECTURE

The CDMU consists of two Processor Modules (PM1 and PM2), two Reconfiguration Modules (RM1 and RM2), and two Telemetry Modules (TMM1 and TMM2). It their own turns, each PM consists of Random Access Memory (RAM), Integer Unit (IU), Floating Point Unit (FPU), and Erasable Electrically Programmable Memory (EEPROM). Each Reconfiguration Module (RM) has two components -- Mass Memory (MM) and On-Board Reference Time (OBRT). Telemetry Modules generate Telemetries (TMs) that are processed by Processor Modules.

In CDMU, only one Processor Module (PM1 or PM2) is in active mode and can access one or both RM1 and RM2. TMs are received by the active processor module and accumulated only in MM of its local RM. However, TMs can be retrieved from the MM of partner RM after switching is done from one processor module to another. When each particular PM has experienced a failure, the Master and Slave policy is introduced for error recovery. It aims at ensuring that the CDMU functionality can be preserved even when failures are present in the system.

In our case study, we consider the following two consecutive errors in CDMU that might occur during the execution of the system:

1) PM1 fails due to the failure in FPU.
2) TM ceases to function due to the failure in the link between TMM2 and PM2.

The basis of the Master and the Slave is to prepare a work-around in order to address above mentioned failures. In this case, PM1 and PM2 are converted into the Slave and the Master respectively. Similarly, Master and Slave comprise of the functional program running in PM2 and PM1 respectively and it is mainly established to execute the system without the FPU and connection link.

At a time, both the Master and the Slave interface with RM1 and RM2, respectively, as shown in the CDMU structure. However, RM1 and RM2 are not capable to hold simultaneous access to both of them.

Despite the error in the connection link of PM2, the PM2 is still in operational mode and stores TM in the MM. Similarly, PM1 is also in operational mode by using only IU program (without FPU) that recovers TM from the MM and sends to the operator. The operator interacts with the Master and the Slave by sending Tele-Commands (TCs). Figure 1 shows that each processor module is connected to both RM1 and RM2 and to both TMM1 and TMM2. The TeleCommand (TC) receiver is also linked to both PM1 and PM2.

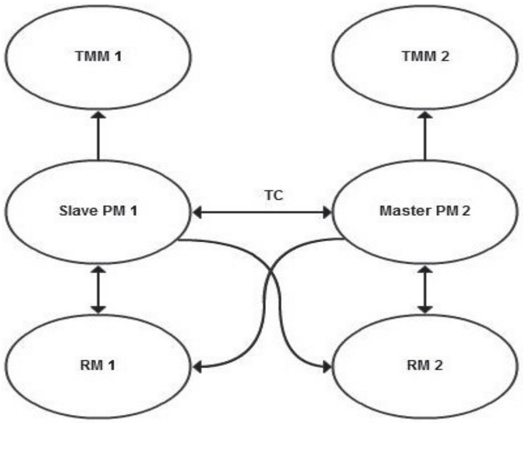


Figure 1: CDMU Structure [1]

IV. FACTORS CONTRIBUTING IN HANDSHAKE

The important key factors that are involved in the handshake procedure are as follows:

1) Time Event Register (TER) is used for messaging between the Master and the Slave. As there is no direct link between the Master and the Slave, so TER is used as a shared device. Both can access TER to read and write messages. RM1 and RM2 have their own TER devices.
2) The two interrupts -- Time Event Interrupt (TEI) and Time Synchronization Interrupt (TSI) caused by RM1 and RM2 are sent to the Slave and the Master respectively. If the Master uses RM1 and interrupt triggers, then interrupt is only sent to the Slave because it is a local processor module of RM1.
3) The interrupts can be used as a signal from the Master to the Slave for the acknowledgement of the messages because the Master has a charge of the interrupt timing.
4) OBRT Status Register is used to find out that interrupt has triggered in the system. The Master holds the check of this register and clears the interrupt flag for allowing the coming up interrupts.
5) The Master and the Slave cannot use the same RM at a time. However, both the Master and the Slave are informed through handshake procedure in order to choose required RM at a given time interval.
6) Handshaking is done through Communication Channel (CCH) between the Master and the Slave. RM1 or RM2 is used as CCH. The TER in the CCH is expressed as Communication Time Event Register (CTER).
7) The selection of RM1 or RM2 as CCH depends on the Master as it utilizes both RM1 and RM2. On getting the TC instruction from the operator, it

switches to one module of RM (RM1 or RM2) and releases the other RM for CCH. If the Master is using only one RM module initially, the unused RM will be selected as CCH. The Master can switch the RM at the end of the handshake procedure.

8) The handshake message contains the phase content and timing of the message that is encoded in the CTER. The timing of the interrupt is slightly affected by the phase content that is encoded in the four Least Significant Bits (LSB) of the CTER, but this affect of interrupt timing is less than 0.3 ms and is, therefore, ignored.
9) The phase content in the four least significant bits of the CTER is as under:
    i. When 4 LSB of CTER has value ‘1’, then the Master informs the Slave to communicate through RM1. Similarly, when 4 LSB of CTER has value ‘2’, then the Master informs the Slave to communicate through RM2. This phase is known as “Select Communication RM”.
    ii. If the value is ‘4’ in the 4 LSB of CTER, the Slave updates the Master to confirm the communication through RM1. Likewise, if the value is ‘5’ in the 4 LSB of CTER, then the Slave informs the Master that it confirms the communication through RM2. This phase of the handshake procedure is called “Confirm Communication RM”.
    iii. Upon setting the value of ‘10’ in 4 LSB of CTER, the Slave is informed by the Master that if RM1 is not in use then switch to it and use it. For the value ‘11’, the Slave has to switch to use RM2. When the value is ‘14’, then the Master instructs the Slave to release both RM1 and RM2. This phase is named as “Command Slave”.
    iv. The Master sends a message to the Slave in which it verifies the RM1 or RM2 selection by putting the value ‘8’ in 4 LSB of CTER. This phase is entitled as “Confirm Command”.

10) The encoding of the handshake messages is done within one second (s) - Pulse Per Second (PPS). The interrupts according to the PPS time slot are given below:
    i. When interrupts occur from 0.10 to 0.40 s, RM1 and RM2 are not selected in this time slot. It means that the Master instructs the Slave to confirm the change to use no RM.
    ii. For the selection of RM1, interrupts take place in the time slot ranging from 0.42 to 0.70 s. The Master orders the Slave either to communicate with RM1 or confirm change to use RM1 during the handshake procedure.
    iii. In the 0.72 - 1.00 s time slot, interrupts are taken into account. This selection is encoded for RM2 where master notifies the Slave either to communicate with RM2 or confirm change to use RM2 during the handshake procedure.
    iv. The purpose of the remaining unused slots 0.00 – 0.10 s, 0.40 – 0.42 s and 0.70 – 0.72 s is to avoid overlaps. Any interrupts appearing in these timing slots will be ignored.

11) The minimum time between two TSIs is greater than 0.3s to ensure that two TSIs do not trigger during the same time slot. On the other hand, interrupt can be triggered two times during the same time slot.

V. Proposed System Model for Handshake

The handshake procedure [2] has been modeled for the Master and the Slave as shown in Figure 2. Handshake is a procedure in which the Master communicates with the Slave to update the selection of RM1 and RM2. It is a complicated process as there is no direct communication link between them.

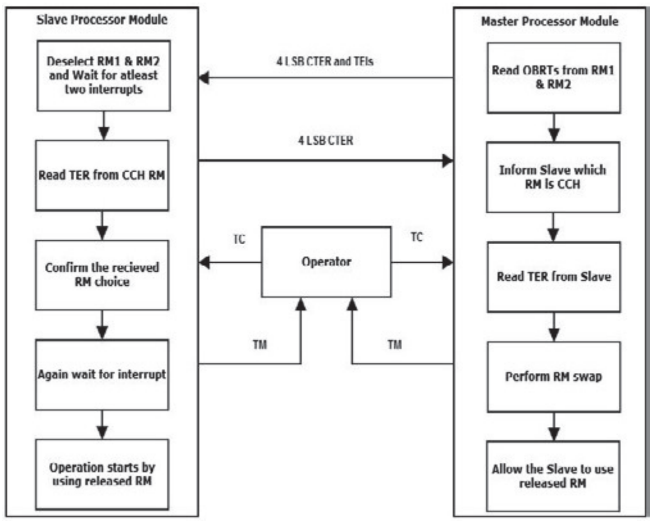

Slave Processor Module
Deselect RM1 & RM2 and Wait for atleast two interrupts
Read TER from CCH RM
Confirm the recieved RM choice
Again wait for interrupt
Operation starts by using released RM
4 LSB CTER and TEIs
4 LSB CTER
TC
TC
Operator
TM
TM
Master Processor Module
Read OBRTs from RM1 & RM2
Inform Slave which RM is CCH
Read TER from Slave
Perform RM swap
Allow the Slave to use released RM

Figure 2: Model of Handshake Procedure

A. *Master Handshake Procedure*

The handshake procedure that is executed by the Master Module is shown in Figure 2. Below we give its brief description:

Upon the reception of TC from the operator, the handshake procedure is started by the Master. The Master informs the Slave that other RM will be used as CCH by updating the value of 4 LSB TER. If the Master is using RM2 and storing TM, then the Slave will be informed to make RM1 as CCH. Likewise, if RM1 is operated by the Master, then the Slave has to use RM2 as CCH. When CCH is RM1, then system operation is performed from 0.42 to 0.70 s PPS slot. Similarly, for RM2, 0.72 to 1.00 s, PPS slot is used for the system operation. System has to wait for starting of the right PPS slot according to the CCH.

In order to send information to Slave, interrupts are triggered from the Master after setting the value of OBRT Status Register to zero. For accuracy, the value of TER for the Slave RM is set to 0.04 s. The interval between two interrupts is 0.06 s. The Master ensures by reading the CTER value from the Slave that selection of CCH is done. The Master can swap the CCH selection at the end of handshake procedure. The Master commands the Slave by setting the future CCH selection value in the 4 LSB CTER and triggers a TEI only. The time value of TEI is not relevant to the CTER, so the time slot of TEI makes no changes in the end result of the system. Only operator is responsible for the new RM selection and determining which RM is used as CCH as stated in Section IV. In the system, operator initially notifies the RM selection to the Master, it changes CCH selection from used RM to other RM according to the swapping information that is encoded in 4 LSB CTER and also confirms the RM selection. The confirmation message is also forwarded to the Slave by sending two interrupts within the correct time slot. At this moment, the Master ends the handshake procedure and updates the operator for successful working by sending the corresponding TM.

B. *Handshake Procedure: Slave Behaviour*

When the operator starts the handshake, the following operations are carried out by the Slave as shown in Figure 2.

If the Slave is using RM1 or RM2, then it will deselect the current RM on the reception of TC command from the operator. When RM is discontinued from the Slave, then OBRT Status Register will be set to zero and no more interrupts will be triggered. The Slave waits for 0.03 s to get the new command along with two interrupts (i.e. TEI and TSI) which will be generated from the Master during the expected PPS slot. When the Slave receives a message from the Master, then it decodes it from the interrupts time slot as mentioned in Section IV (para # 10). For verification, the Slave also interprets the value of 4 LSB CTER as described in Section IV (para # 9). If the values derived from the interrupts time slot and 4 LSB CTER are the same, then the Slave achieves the specified CCH selection. After that, the Slave sends acknowledgement of confirmation to the Master by setting the value of 4 LSB CTER according to Section IV. Now, the Slave has to wait again for 0.02 s for the new response or interrupt from the Master according to the PPS slot. On the arrival of message from the Master, the Slave is triggered by TEI. The Slave has no opportunity to change the decision of new selection and waits for 10s for the confirmation message from the Master. Again, the Slave receives two interrupts with the CTER message and compares the time slot of interrupts with previous CTER value. If both are same, then the Slave begins the operation with released RM. Finally, the Slave also completes the handshake procedure by sending TM to the operator.

VI. VERIFICATION OF THE HANDSHAKE MODEL

The handshake model has been implemented by using PROMELA (PROcess MEta LAnguage) high level modeling language with SPIN model checker for verifying the required results. SPIN [3,4] is extensively used in formal verification of distributed and parallel processing systems. SPIN has greatly facilitated the process of verification in the areas of mission-critical algorithmic applications, message and data communication in the client-server environment, synchronization and coordination of large number of processes in the parallel and distributed systems, deadlock handling methodologies in the modern multi-tasking operating systems, verification of the mission-oriented control models for space aircrafts, utilization of intelligent models for determining most suitable and economical paths over wide area networks, checking performance of routing protocols [5], testing of fault-tolerant strategies and implementation of a wide variety of switching techniques. The literature review reveals that most of the software-based systems/models are checked and verified by the SPIN model checker.

The handshake model between two processors in control and data management unit has been successfully implemented and verified using SPIN/PROMELA. The flow chart for handshake procedure model is shown in Figure 3. The following algorithm along with description of each condition of the processes shows part of the implemented SPIN/PROMELA model.

```
/*Variable Declarations */
active proctype Slave_starts_HP()
{S_TC=true;
if
::(S_TC==true)->RM1=0;RM2=0;
::( S_TC!=true)-> printf("\n\nExit Handshake Procedure.\n\n");
fi
S_TM=true;}
```

The above code depicts that when TC command is received to Slave from the operator, Slave starts handshake procedure by deselecting the RM selection. After successful execution of the TC command, Slave sends TM to operator and waits for Master's response. In any other condition, handshake procedure will be terminated.

```
active proctype Master_starts_HP()// time value is taken in (ms)
{M_TC=true; RM1=0;RM2=1; // set by the operator
if
::(RM1==0 && RM2==1)->// I_time denotes timing of interrupts
{CTER_4_LSB=1;I_time=500;TEI=true;TSI=true;OBRT_SR=1;
run Slave_read_wrtie_operation(CTER_4_LSB,I_time,TEI,TSI);}
::(RM1==1 && RM2==0)->
{CTER_4_LSB=2;I_time=800;TEI=true;TSI=true;OBRT_SR=1;
run Slave_read_wrtie_operation(CTER_4_LSB,I_time,TEI,TSI);}
fi}
```

The code associated with the above process describes that Master starts handshake on the operator command. When operator selects RM2 for Master, then Master uses RM2 and notifies Slave (by sending CTER and interrupts) to use RM1 as CCH. Likewise, if operator selects RM1, then Master uses RM1 and updates the Slave (through CTER and interrupts) to use RM2 as CCH. After that, it waits for Slave's response.

```
proctype Slave_read_wrtie_operation(int CTER_4_LSB,I_time;bool
TEI,TSI)
{if
::((CTER_4_LSB==1) && (TEI==true && TSI==true) && (I_time>=420
&& I_time<=700))->
{CTER_4_LSB=4;run Master_decides_future_selection(CTER_4_LSB);}
::((CTER_4_LSB==2) && (TEI==true && TSI==true) && (I_time>=720
&& I_time<=1000))->
{CTER_4_LSB=5;run Master_decides_future_selection(CTER_4_LSB);}
::((CTER_4_LSB!=1) || !(I_time>=420 && I_time<=700))->
{printf("\n\nExit Handshake Procedure.\n\n");}
::((CTER_4_LSB!=2) || !(I_time>=720 && I_time<=1000))->
{printf("\n\nExit Handshake Procedure.\n\n");}
fi}
```

The above piece of code illustrates that when timing of interrupts is in line with the information that is encoded in CTER 4 LSB, then Slave confirms the selection to Master and waits for 0.02 s in order to get Master's response. So, when interrupts occurs between 0.42 to 0.70 s time slot and CTER 4 LSB is '1', it means Slave confirms to use RM1 as CCH by encoding the value '4' in CTER 4 LSB. Similarly, if time slot for interrupt is 0.72 to 1.00 s and CTER 4 LSB is '2' then RM2 is confirmed as CCH by the Slave through updating the value '5' in CTER 4 LSB. If timing of the interrupts is not compatible with the encoded information in CTER 4 LSB, handshake procedure exits at this stage.

```
proctype Master_decides_future_selection(int CTER_4_LSB)
{if
::(CTER_4_LSB==4)->
{OBRT_SR=0;CTER_4_LSB=11;TEI=true;OBRT_SR=1;
if
::(CTER_4_LSB==11)->
{RM1=1;RM2=0;aa= CTER_4_LSB;OBRT_SR=0;CTER_4_LSB=8;
I_time=800;TEI=true;TSI=true;OBRT_SR=1;M_TM=true;
run Slave_interprets_message(aa,I_time,TEI,TSI);}
::(CTER_4_LSB==14)->
{RM1=0;RM2=0;OBRT_SR=0;aa=CTER_4_LSB;CTER_4_LSB=8;
I_time=200;TEI=true;TSI=true;OBRT_SR = 1;M_TM=true;
run Slave_interprets_message(aa,I_time,TEI,TSI);}
fi;}
```

The above fragment of the code describes that when Slave is using RM1, Master updates the up-coming selection of RM by placing the value '11' or '14' in CTER 4 LSB with only TEI. If Master selects RM1, it releases RM2 to be used as CCH by putting the value '11' in CTER 4 LSB. When Master picks RM1 and does not release RM2 to be used as CCH, it writes the value '14' in CTER 4 LSB. After a half second to give the Slave sufficient time to read value of CTER, the Master confirms the selection to the Slave by encoding the value '8' in CTER 4 LSB on the specified time slot according to Section IV and exits the handshake procedure.

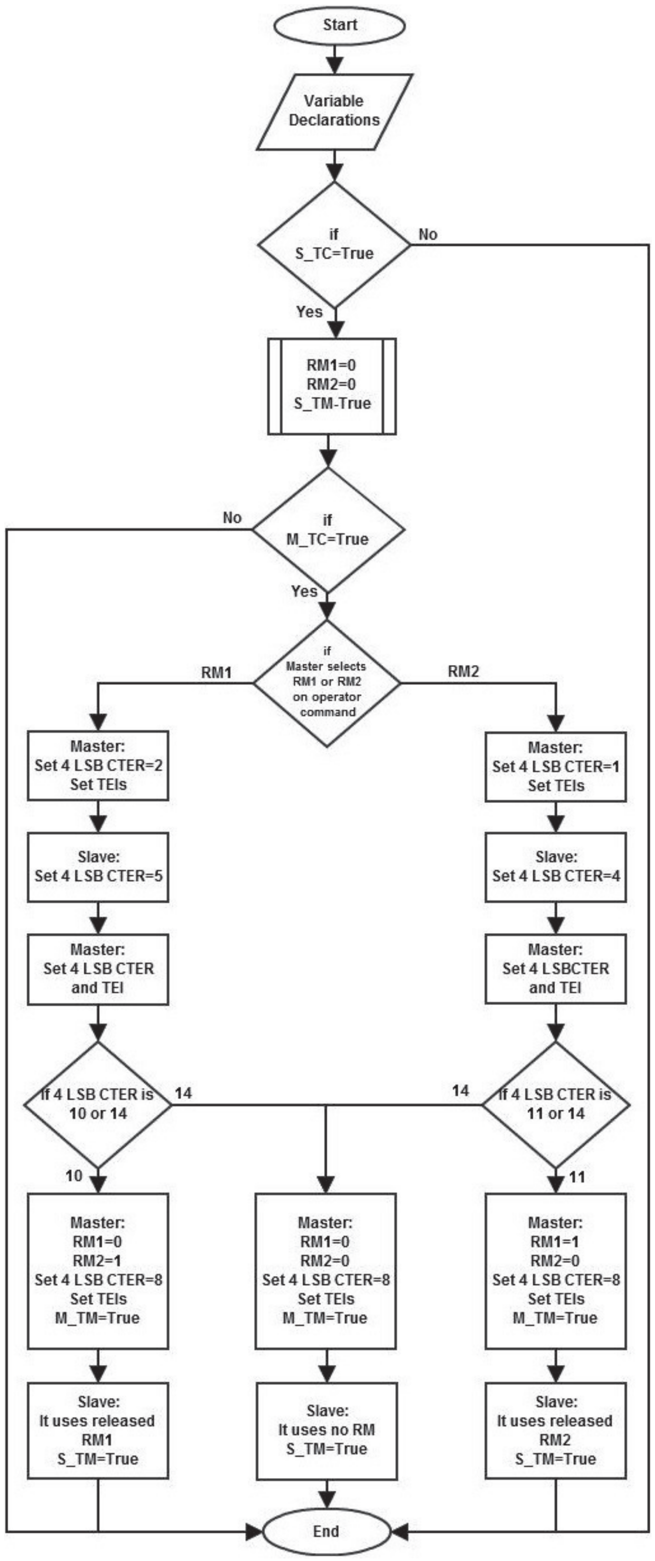

Start
Variable Declarations
if S_TC=True
No
Yes
RM1=0
RM2=0
S_TM-True
if M_TC=True
No
Yes
if Master selects RM1 or RM2 on operator command
RM1
RM2
Master: Set 4 LSB CTER=2 Set TEIs
Master: Set 4 LSB CTER=1 Set TEIs
Slave: Set 4 LSB CTER=5
Slave: Set 4 LSB CTER=4
Master: Set 4 LSB CTER and TEI
Master: Set 4 LSBCTER and TEI
If 4 LSB CTER is 10 or 14
14
14
If 4 LSB CTER is 11 or 14
10
11
Master: RM1=0 RM2=1 Set 4 LSB CTER=8 Set TEIs M_TM=True
Master: RM1=0 RM2=0 Set 4 LSB CTER=8 Set TEIs M_TM=True
Master: RM1=1 RM2=0 Set 4 LSB CTER=8 Set TEIs M_TM=True
Slave: It uses released RM1 S_TM=True
Slave: It uses no RM S_TM=True
Slave: It uses released RM2 S_TM=True
End

Figure 3: Flow Chart of Handshake Procedure Model

```
::(CTER_4_LSB==5)->
{OBRT_SR=0;CTER_4_LSB=10;TEI=true;OBRT_SR=1;
```

The associated code with above condition illustrates this.

```
if
::(CTER_4_LSB==10)->
{RM1=0;RM2=1;OBRT_SR=0;bb=CTER_4_LSB;CTER_4_LSB=8;
I_time=800;TEI=true;TSI=true;OBRT_SR=1;M_TM=true;
run Slave_interprets_message(bb,I_time,TEI,TSI);}
::(CTER_4_LSB==14)->
{RM1=0;RM2=0;OBRT_SR=0;bb=CTER_4_LSB;CTER_4_LSB=8;
I_time=200;TEI=true;TSI=true;OBRT_SR = 1;M_TM=true;
run Slave_interprets_message(bb,I_time,TEI,TSI);}
fi;}
fi}
```

The above part of the code shows that when the Master is using RM1, it updates the up-coming selection of RM by setting the value '10' or '14' in CTER 4 LSB with only TEI. If the Master selects RM2, it releases RM1 to be used as CCH by putting the value '10' in CTER 4 LSB. When the Master picks RM2 and does not release RM1 to be used as CCH, it writes the value '14' in CTER 4 LSB. After a half second to give the Slave sufficient time to read value of CTER, the Master confirms the selection to the Slave by encoding the value '8' in CTER 4 LSB on the specified time slot according to Section IV and exits the handshake procedure.

```
proctype Slave_interprets_message(int previous_CTER,I_time;bool
TEI,TSI)
{if
::((I_time>=420 && I_time<=700) && (previous_CTER==10) &&
(TEI==true && TSI==true))->
{S_TM=true;}
::((I_time>=720 && I_time<=1000) && (previous_CTER==11) &&
(TEI==true && TSI==true))->
{S_TM=true;}s
::((I_time>=100 && I_time<=400) && (previous_CTER==14) &&
(TEI==true && TSI==true))->
{S_TM=true;}
::(!(I_time>=420 && I_time<=700) || (previous_CTER!=10))->
{ printf("\n\nExit Handshake Procedure.\n\n");}
::(!(I_time>=720 && I_time<=1000) || (previous_CTER!=11))->
{ printf("\n\nExit Handshake Procedure.\n\n");}
::(!(I_time>=100 && I_time<=400) || (previous_CTER!=14))->
{ printf("\n\nExit Handshake Procedure.\n\n");}
fi}
init
{atomic// Atomic is used to reduce the complexity.
{run Slave_starts_HP();
run Master_starts_HP();}
}
```

The code given above indicates that after waiting for 10 s, Slave receives the confirmation message with two interrupts from Master. The timing of interrupts is matched with the information that is encoded in previous CTER 4 LSB as mentioned in Section IV. Therefore, when timing of the interrupts lies between 0.42 to 0.70 s time slot and previous CTER 4 LSB is '10', it notifies that Slave uses RM1 as CCH that is released by the Master. Similarly, timing of the interrupts lies between 0.72 to 1.00 s time slot and previous CTER 4 LSB is '11', it notifies that Slave uses RM2 as CCH that is released by the Master. Also, when interrupts timing lies between 0.10 to 0.40 s and the value of previous CTER 4 LSB is '14', then Slave uses neither RM1 nor RM2 as CCH. After then Slave exits the handshake procedure. If interrupts timing is not in line with the information that is encoded in earlier CTER 4 LSB, handshake procedure exits at this stage too.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed a formal approach for modeling a fault-tolerant satellite system that relies on the handshake procedure for dynamic reconfiguration. We have demonstrated how to create a Promela model of the handshake and carry out its analysis. Since the handshake procedure has a number of non-trivial properties caused by the distributed nature of the system, such a model allows the designers to ensure correctness of the handshake implementation. In our future work, we are planning to extend the proposed approach to derive the generic modeling patterns. Moreover, it would be interesting to explore the handshake in the presence of more complex network architecture.

REFERENCES

[1] "DEPLOY – Software Requirement Specification, Master/Slave Software", Space Systems Finland, Ltd., July 2011.

[2] J. Kashif, and E. Troubitsyna, "Designing a Fault-Tolerant Satellite System in SystemC", ICONS 2012, The Seventh International Conference on Systems, IEEE Computer Press, pp. 49–54, March 2012.

[3] C.Baier and J.-P. Katoen. "Principles of Model Checking". MIT Press, 2008.

[4] N. A. S. A. Larc, "What is Formal Methods?", NASA Langley Methods, http://shemesh.larc.nasa.gov/fm/fmwhat.html, formal methods program, 2001.

[5] J. Kashif, A. Kashif, and E. Troubitsyna,, "Implementation of SPIN Model Checker for Formal Verification of Distance Vector Routing Protocol", International Journal of Computer Science and Information Security (IJCSIS), Vol 8, No 3, USA, ISSN 1947-5500, pp. 1-6, June 2010.

[6] A. Tarasyuk, I. Pereverzeva, E. Troubitsyna, T. Latvala, and L. Nummila, Formal Development and Assessment of a Reconfigurable On-board Satellite System, In: Frank Ortmeier, Peter Daniel (Eds.), Proceedings of 31st International Conference on Computer Safety, Reliability and Security (SAFECOMP 2012), LNCS 7612, pp.210-222, Springer, 2012.

Paper V.

E. Troubitsyna and K.Javed, “Towards Systematic Design of Adaptive Fault Tolerant Systems”, ADAPTIVE 2014, The Sixth International Conference on Adaptive and Self-Adaptive Systems and Applications, pp. 15-21, May 2014, Venice, Italy.

Towards Systematic Design of Adaptive Fault Tolerant Systems

Elena Troubitsyna, Kashif Javed
Åbo Akademi University, Finland
e-mails: {Elena.Troubitsyna, Kashif.Javed}@abo.fi

Abstract—**The development of modern distributed software systems poses a significant engineering challenge. The system architecture should exhibit plasticity and high degree of reconfigurability to enable an automated adaptation to continuously changing operating conditions and component failures. Traditional engineering approaches are inefficient to cope with complexity of such systems to ensure their robustness and fault tolerance. Therefore, there is a clear need for the approaches explicitly addressing the problem of designing adaptive fault tolerance mechanisms. In this paper, we propose a systematic approach to the development of adaptive fault tolerant systems. We discuss the main principles of architecting such systems to enable plasticity and reconfigurability. We demonstrate how deployment of the predictive adaptation allows us to ensure that the system would be able to continuously deliver its services with the acceptable quality despite occurrence of component failures.**

Keywords-adaptable systems; fault tolerance, predictive adaptation; reconfiguration.

I. INTRODUCTION

The complexity of modern large-scale systems requires solutions that ensure that systems autonomously adapt to the operating environment and internal conditions. Often, such systems are put into a wide class of autonomic systems -- the software-intensive systems that, besides providing their intended functionality, are also capable to diagnose and recover from errors caused either by external faults or unforeseen state of environment in which the system is operating [3]. In this paper, we focus on the fault tolerance aspect of such systems.

Fault tolerance is an ability of a system to deliver its services in a predictable way despite faults [8]. The generic principle underlying design of fault tolerant systems is to detect a discrepancy between a model representing fault free system behaviour and the observed state, and implement error recovery [8] .

In this paper, we propose a general pattern for architecting and developing the adaptive fault tolerant systems. The proposed pattern supports a layered design approach [6] that enables separation of concerns and facilitates structured design of fault tolerance mechanisms. In our representation of the architectural pattern, we define the interfaces between the components at different levels of abstraction to ensure correct propagation of fault tolerance related data. The high-level coordination of the fault tolerance mechanisms is implemented by an adaptation manager – a component that is responsible for implementing predictive fault tolerance. To specify the adaptation manager, we propose an algorithm that allows the adaptation manager to monitor state of the system at the run time and implement proactive adaptation. Such an approach ensures that the overall system would continuously deliver the services with the acceptable quality. We believe that the proposed approach ensures a systematic development of adaptive fault tolerant systems.

The paper is structured as follows: in Section II, we overview the state-of-the-art in designing adaptive fault tolerant systems. In Section III, we describe general principles of achieving fault tolerance, and, in particular, proactive fault tolerance. In Section IV, we present our proposal for structuring adaptive fault tolerant system. In Section V, we present our proposal for algorithms that implement proactive fault tolerance. Finally, in Section VI, we discuss the proposed approach and future work.

II. RELATED WORK

The need for high performance and continuous service provisioning demands novel solutions for achieving system fault tolerance. We are increasingly observing deployment of proactive fault tolerance techniques that replace traditional reactive approaches [10]. In modern large-scale systems, error rate is increasing and reliance on traditional “error-detection – error-recovery” pattern leads to poor performance and prolonged system downtime, which is often unacceptable. The approaches for proactive fault tolerance are based on preventive treatment of faults aiming at precluding failures and minimising recovery time [10]. The main mechanism of achieving proactive fault tolerance is adaptation.

The problem of software adaptation has been extensively studied at the implementation level, (see e.g., [2] for an overview). However, there is a lack of approaches that attempt to derive appropriate adaptation mechanisms from system-level goals as well as support layered reasoning needed to efficiently cope with system complexity. A prominent work on formal modelling of adaptive systems has been done within the HATS project [2]. In [13][14], an approach to quantitative assessment of reconfiguration strategy has been proposed. In our previous work, we also investigated the impact of faults on dependability, as well as structured approach to designing fault tolerant distributed systems [7][11].

Current engineering practice takes an architecture-centric perspective on adaptive systems. Among the most prominent examples are the Rainbow framework proposed at Carnegie Mellon University [12] and the autonomic computing initiative by IBM [3]. These frameworks outline the main abstractions for describing and managing dynamic system changes. However, currently, the approaches to proactive fault tolerance are not well-integrated into the system development process [10]. In this paper, we will address this problem by proposing a structured approach to architecting adaptive fault tolerant systems. Our approach aims at facilitating design space exploration at the early development stages and enabling explicit representation of the mechanisms for proactive fault tolerance.

III. FAULT TOLERANCE

The main goal of introducing fault tolerance is to design a system in such a way that faults of components do not result in a system failure. A fault cannot be detected by a system until the manifestation of the fault generates *errors* in the component function. The first step in implementing fault tolerance is error processing [10]. *Error processing* aims at removing errors from the computational state.

The first step in error processing is *error detection*. An error is a manifestation of a fault. The general mechanism of error detection is to intercept outputs produced by a system (or a component) and to check whether those outputs conform to the specification of fault free behaviour. Discrepancy between produced outputs and the specification indicates an occurrence of an error. The next step in error processing – *damage confinement* – is concerned with structuring the system to minimise the spread of errors. Once the damage is assessed and confined the error recovery can be performed. *Error recovery* has two main forms – forward and backward error recovery. The forward error recovery mechanisms manipulate the current system state to produce a new system state, which is presumably error free. The success of error recovery strongly depends on how precisely the error is located and how well it is confined. A typical example of forward recovery is *failsafe* [1]. If a system has a safe though non-operational state then it may be possible to recover from an error by forcing the system permanently to that safe state (obviously, this strategy is only appropriate where shut down of the system operation is possible).

By analyzing actions to be undertaken for error processing, we observe that error processing imposes additional requirements on the system design. Namely:

- The system should be specified in such a way that error occurrence conditions are easily deduced and then explicitly checked;
- The system architecture should enable error confinement;
- Error recovery procedures should be identified for every output, which differs from the specified one.

Obviously, an incorporation of error processing in the system design has a strong impact on all levels of the system structure. Hence, fault tolerance should be an intrinsic part of system development and should start from the early stages of the system design.

To embrace complexity challenge, fault tolerance community has been proposing new concepts that can be seen from initiatives and research efforts on autonomic computing [3] and various forums on self-healing [9] or self-protection (see, e.g., [1]). These terms span a wide range of research fields ranging from adaptive memory management to advanced security mechanisms.

A promising direction among them focuses on determining how computer systems can proactively handle failures: if the system knows about a critical situation in advance, it can try to apply countermeasures in order to prevent the occurrence of a failure, or it can prepare repair mechanisms for the upcoming failure, in order to reduce the time-to-repair.

Such an approach can be called proactive fault tolerance. It encompasses three main steps:

1. Failure prediction: it aims at identifying failure-prone situations, i.e., the situations that will probably evolve into a failure. The result of failure prediction is an evaluation of whether the current situation is failure-prone.

2. Proactive reconfiguration: based on the outcome of failure prediction, a system should make a decision and implement the countermeasures to be executed in order to remedy the problem. These decisions are based on an objective function taking into account the cost of the actions, the confidence in the prediction, and the effectiveness and complexity of the actions to determine the optimal tradeoff. Challenges for action execution include online reconfiguration of globally distributed systems, data synchronization of distributed data centers, and many more.

3. Recovery: this stage enables graceful degradation of services while the resources are insufficient for mitigating the failures. For instance, the predictive reconfiguration might not be completed as promptly as expected and the system should compensate for insufficient resources. Another example would be a sudden simultaneous failure of several components due to unexpectedly adverse situations in the environment.

Each one of these stages is important for an efficient implementation of the proactive fault tolerance. Hence, novel architectural solutions, algorithms and development approaches are needed to attain the goal of building adaptive fault tolerant systems.

To build a proactive fault tolerance solution that is able to boost system dependability, the best techniques from all fields for the given surrounding conditions have to be combined.

In this paper, we consider the proactive fault tolerance to be the main adaptation mechanism to achieve system dependability. In the next section, we present our approach to structuring an adaptive fault tolerant system. Then, we focus on designing the proactive adaptation mechanisms. Our proposal aims at enhancing self-adaptation system capabilities. Our goal is to design the mechanisms that allow a system to autonomously adapt to changing operating conditions without human intervention. Essentially, our proposal follows a spirit of the autonomic computing paradigm.

IV. Architecture of Adaptive Fault Tolerant Systems

In this paper, we propose to structure an adaptive fault tolerant system in a layered manner [6]. The layered architecture significantly simplifies the development of complex software-intensive systems. Each layer becomes responsible for a certain aspect of the system behaviour. It facilitates a clear separation of concerns and simplifies the interfaces between the layers. The main issue is to device a well-structured clean architecture that does not introduce tangled interdependencies between layers. In this paper, we propose to structure the architecture of a fault tolerant adaptive system in four layers:

- Application layer
- Adaptation layer
- Fault tolerance layer
- Physical layer

The physical layer represents the environment whose state should be monitored. It might be a complex control system that uses sensors to monitor the health of its components. Another example might be an indoor sensor network that monitors such conditions as temperature, humidity, the level of CO, etc. Finally, it might also be a sensor network for monitoring the outdoor environment, e.g., such as used for forest fire detection, air pollution etc.

The fault tolerance layer performs the data aggregation and evaluation of the quality of monitoring. This information is supplied to the adaptation layer that is responsible for defining the proactive adaptation policy. The aim of the application and fault tolerance layer is to continuously supply the application with the monitoring data of an acceptable quality. The design of the application is defined by its purpose – it varies from the complex control functions to collecting data intelligence. The graphical representation of the system architecture is given in Fig.1.

The physical layer consists of the component to be controlled by the application software. In order to implement proactive fault tolerance, the software should continuously monitor the state of the controlled components.

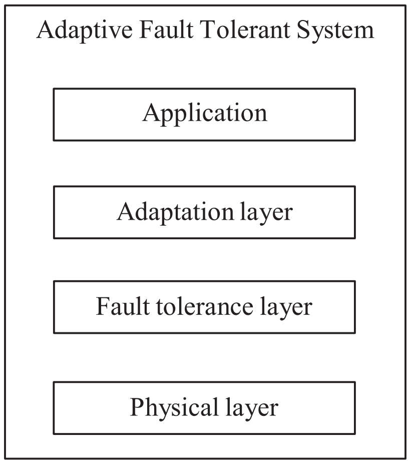


Figure 1. Structure of an adaptive fault tolerant system.

The monitoring capabilities are achieved by integrating sensors that measure the parameters required to observe the behaviour of the system in real-time. Usually, complex systems contain a large number of sensors. Hence, from the fault tolerance perspective, the physical layer can be considered as a sensor network.

It generates raw data. Each sensor produces the data in the following format

<value, timestamp>

We consider two most typical failure modes of the sensors: *stuck at previous value* and producing a (detectably) *incorrect value*. In the former case, the sensor fails silently by failing to update its reading, i.e., the timestamp indicates that the produced data is old. In the latter case, the sensor produces the value that is outside of the feasible range.

At the fault tolerance layer resides fault tolerance manager. The goal of the fault tolerance manager is

- To periodically read the sensor data,
- To filter out faulty data,
- To compute the average value of valid data together with defining the quality level.

The fault tolerance manager produces the input for the adaptation manager as a tuple

<value, level>

To compute the quality level, the fault tolerance manager keeps track of the number of sensors that have produced valid data. There are two thresholds: *lim1* and *lim2* such that *lim2 > lim1*. They determine the quality level. If the number of the sensors that produced the valid data is greater than *lim2* then the quality level is set to *Level 3*. If the number of sensors produced valid data is between *lim1* and *lim2* then the quality level is set to *Level 2*. If the number of valid readings is between *1* and *lim1* then the quality level is set to *Level 1*. Finally, if none of the sensors have produced valid results then the quality level is assigned value *Level 0*.

The adaptation manager and deployment manager constitute the adaptation layer. The adaptation manager receives the data from the fault tolerance manager in the format

<*value, level*>

where *level* is an integer between *0* and *3*. If the level has value *3,* then, the value has a good quality and the adaptation manager simply forwards the received value to the applications. However, if the quality level is below *3* but greater that *0* then the adaptation manager still forwards the received data to the application but starts an observation period.

The aim of the observation period is to establish whether the decline in the quality of data is temporal or permanent. Assume that, after receiving a value with the levels *1* or *2,* the adaptation manager observes a continuous period of receiving data with quality level *3*. Then, the observation period terminates and no reconfiguration is initiated, i.e., the adaptation manager treats the decline in the quality of data as a temporal one and considers the system to be healthy.

If, during the observation period the adaptation manager continuously receives data with quality level *1* or *2* then after the observation period expires, it initiates reconfiguration, i.e., considers the quality deterioration to be the permanent one.

The reconfiguration is triggered by sending a request to the *deployment manager* to deploy a new set of sensors. The deployment can be achieved in several different ways. For instance, if we consider a wireless sensor network that is used to monitor the state of the environment then the deployment is performed via a distribution of a set of fresh sensors (e.g., from an airplane). If the sensors are used to monitor an indoor environment then the deployment triggers a request to the maintenance company. The same principle applies if the sensor network is used to monitor the behaviour of a complex control system. In any case, the main advantage of the proposed approach is a possibility to preventively react on the deterioration of the quality of monitoring and avoid the loss of the observability of the physical layer.

The requested number of new sensors to be deployed depends on how deeply the level of data quality has deteriorated. If the quality level has value *1* then the deployment manager requests *n* new sensors to be deployed. If the quality level has the value *2* then *m* new sensors are to be deployed, where $m<n$.

In general, we could design a more sophisticated deployment mechanism. For instance, if each sensor or a group of sensors is assigned an id then the failures can be diagnosed precisely. This would allow the adaptation manager to communicate the exact requirements for the deployment of new sensors.

When the new sensors are deployed, the deployment manager acknowledges the completion of the reconfiguration and the adaptation manager notifies the fault tolerance manager about availability of the new sensors. The fault tolerance manager closes the connection with the failed sensors and establishes connection with the newly deployed ones.

An important aspect to be considered is how to define the behaviour of the adaptation manager when the quality level keeps fluctuating between the values 2 and 3. On the one hand, the adaptation manager should not trigger the reconfiguration prematurely. On the other hand, delaying a reaction on such an unstable situation might result in an abrupt deterioration of the quality of data that should be prevented.

To resolve this issue, we let the adaptation manager to maintain the observation period as long as no continuous improvement in quality has been observed. Every time when the data are received with the quality threshold lower than *3*, the adaptation manager increments the counter of the observation period. When this counter exceeds the predefined threshold, the adaptation manager triggers the reconfiguration. This approach is taken to ensure that the preventive reconfiguration will be initiated even if the system keeps fluctuating between quality levels.

Finally, if the adaptation manager receives data with the quality level equal to *0,* then it immediately initiates reconfiguration of the data flow. In this case, it starts to send to the application data received at the previous cycle. It continues to send the last data with an acceptable quality value until the reconfiguration is completed and the fault tolerance manager starts to send the data with an acceptable quality level.

In the next section, we define the main behavioural patterns of adaptation manager and fault tolerance manager.

V. ALGORITMS FOR PROACTIVE FAULT TOLERANCE

Let us focus first on defining the module specifying the fault tolerance manager.

The module should implement the procedures of

- Reading the sensor data,
- Checking validity of sensor data with respect to time and feasibility
- Calculating the average of the received valid data and the quality level.

In our definition of the fault tolerance manager, we used two abstract functions *fresh* and *valid*. The function *fresh* relies on the specific parameters to determine whether the produced data is fresh. Since the clocks of the sensors might fluctuate, the function checks whether the timestamp is within certain boundaries.

The function *valid* checks feasibility of the data produced by a sensor. It returns the Boolean value *True* if the data is valid and *False* otherwise.

```
Module Fault Tolerance Manager
Global Variables
   in_buffers: array of  <float, INT>
   out_buffer: seq of <float, INT>

Local Variables
   count: INT /*counter of healthy sensors
   sum : float /*sum of readings
   avg: float /*average value
   level:  [0..3]

Initialisation:
   count:= 0;
   sum:= 0;
   avg:= 0;
   level:= 0

Begin

 for i = 1 to k do
   read (data, time_stamp, in_buffer[i]);
   if
        fresh (time_stamp) = True & valid(data)= True
    then  count:= count +1; sum := sum +data
   end;

 if counter > 0 then avg:= sum/count;

 case count = 0 then level:= 0
  elseif count>0 & count<lim1 then level:=1
  elseif count>lim1 & count<lim2 then level:=2
  else level:=3;

 out_buf:= out_buf^<avg,level>;
 count:= 0;
 sum:= 0;
 avg:= 0

End
```

Figure 2. Fault Tolerance Manager.

Reliance of the abstract functions allows us to parameterise the definition of the module and reuse the proposed definition in different contexts.

In our definition of the module, we have abstracted away from the implementation details of the communication between the fault tolerance manager and the sensors. We assume that they communicate by shared variables -- data and time stamps that are stored in the *in_buf* array of pairs.

The proposed algorithm implements the procedure of reading the sensor data, checking their validity with respect to time and feasibility and calculates the average of the received valid data.

By keeping track of the number of valid readings, the fault tolerance manager calculates the quality level. It compares this number with two constants – *lim1* and *lim2*. The pair of calculated data and the quality level is appended to the output buffer that is read by the Adaptation Manager. The specification of the Fault Tolerance Manager module is given in Fig. 2 and the Adaptation Manager in Fig. 3.

```
Module Adaptation Manager

Global Variables
     a_out_buf: float

Local variables:
     observ : Bool
     cur_level : INT
     cur_data:float
     fault_count : INT
     suc_count : INT
     mode: {Normal, Adapt, Adapt_Compl, Adapt_activ}

Initialisation:
     observ :=0;
     cur_level :=0;
     fault_count :=0;
     suc_count :=0;

Begin

 cur_level, cur_data := head(out_buf);

if observ= False & cur_level= 3 then out_buf:= cur_data

if observ= False & cur_level= 2 & fault_count<thr
 then fault_count:= fault_count+1; out_buf:= cur_data;

if observ= False & cur_level<3 & cur_level>0 &
   fault_count>thr-1
 then mode := adapt_active, adapt_req:= True;

if observ= False & cur_level=3 & fault_count>0 & fault_count<thr-
   1
then observ:= True; suc_count := suc_count +1;
     observ:= 0;

if observ= True & cur_level=3 & fault_count>0 &
  fault_count<thr-1 & suc_count<thr_s
then suc_count:= suc_count+1;
     observ_s_iter:= observ_s_iter:=+1;

if observ= True & cur_level=3 & fault_count>0 &
  fault_count<thr-1 & suc_count>thr_s-1 &
  suc_count =observ_s_iter
then observ:= False ; suc_count:= 0; fault_count:= 0;
      observ_s_iter:= 0;

if observ= True & cur_level<3 & fault_count>0 &
   fault_count<thr-1 & suc_count<thr_s
then suc_count:= suc_count+1;
    observ_s_iter:= observ_s_iter+1;

if mode= adapt_activ then adapt_req ;

if adapt_conf  then mode:= normal
End
```

Figure 3. Adaptation Manager.

In the specification of the Adaptation manager, the variable *observ* indicates whether the observation period has started. The variable obtains the value *True* when the first data with the quality level below *3* is received. The variable is reset to *True* if the quality has recovered or a new period of observation is initiated.

The variables *cur_level* and *cur_data* designate the data and the quality level received from the fault tolerance manager. The variable *fault_count* is used to keep track of the number of iterations, in which the data with the quality level lower than 3 have been received. When the value of *fault_count* exceeds the predefined threshold *thr*, the reconfiguration is triggered.

The variable *suc_count* is used to keep track of the iterations that produced data with the quality level 3 after the observation period has been initiated. When the value of *suc_count* exceeds the predefined threshold *thr_s* the adaptation manager has continuously received the data with the quality level 3 for sufficiently long period of time. Therefore, the quality level has recovered and the observation period can be deactivated.

The adaptation manager provides the application with the latest data by updating the global variable *a_out_buf*. It forwards the data received from the fault tolerance manager if the quality level is higher than zero. Otherwise, it simply does not update the variable.

The adaptation manager triggers the reconfiguration by issuing the adaptation request *adapt_req* that is received by the deployment manager. When the new sensors are deployed the deployment manager confirms the reconfiguration by issuing the signal *adapt_conf*.

After triggering the reconfiguration, the adaptation manager enters the mode *Adapt*. After the reconfiguration is completed, the adaptation manager enters the mode *Adapt_Compl*. In this mode [4] [5], it notifies the fault tolerance manager about availability of new healthy sensors. As a response to this, the fault tolerance manager shuts down the connection with the failed sensors and establishes a new connection with the newly deployed sensors. After this procedure is completed, the fault tolerance manager notifies the adaptation manager. It enables transition to the mode *Normal*.

The general scheme of an implementation of the mode transition is given in Fig. 4. The main principle that underlies the mode transition is as follows: the mode is stable and unchanged until a fluctuation in the quality level is registered. We show the snippet implementing this principle as a generic mode changing procedure.

The proposed architecture ensures a separation of concerns and clear allocation of responsibilities between the components. Indeed, the fault tolerance manager is responsible for collecting data and validating them. It encapsulates the failures of sensors and gives only the high-level indication of the current health of the system by annotating the data with the quality level. The adaptation manager is responsible for diagnosing the situation and executing the preventive reconfiguration – requesting the new sensors to be deployed before the quality of data deteriorates below the acceptable level. At the same time, it also ensures remedial actions when no data is produced – it outputs to the application the last healthy value. Such behaviour ensures graceful degradation of quality of service.

Procedure *ModeTransition*

```
Variables
  last_mode: {Normal, Adapt, Adapt_Compl, Adapt_activ}
  next_target: {Normal, Adapt, Adapt_Compl, Adapt_activ}
  prev_target: {Normal, Adapt, Adapt_Compl, Adapt_activ}
  level: int

Begin

if adaptation completed
then initiate a forward transition
     to next_target according to
     the predefined scenario;

if level dropped
then initiate a backward transition to next_target
     adaptation mode
    The choice of target mode depends on severity
    of level decrease;

 if the conditions for entering the target
   mode are satisfied
then complete a transition to next_target mode
     and become stable ;

if neither the conditions for entering
 the next global mode are satisfied nor the level dropped
then maintain the current mode

End
```

Figure 4. Mode transition procedure.

VI. Conclusion and Future Work

In this paper, we have proposed a systematic approach to architecting adaptive fault tolerant systems. We have demonstrated how to structure the system to facilitate layered design of proactive fault tolerant mechanisms. We defined the information flow between the layers of the system architecture that enables adaptation and guarantees a continuous delivery of services with an acceptable quality level.

Proactive fault tolerance is a promising research direction that aims at providing systems with capabilities of executing preventive reconfiguration to preclude occurrence of failure and disruption in service provision. In our paper, the main mechanism of achieving proactive fault tolerance relies on several levels of error detection and monitoring of system health.

As a future work, we are planning to investigate alternative approaches to preventive reconfiguration as well as conduct quantitative assessment of various system characteristics, e.g., correlation between frequency of the network rejuvenation with new sensors and quality of data, proportion between periods of low quality data and different thresholds etc. Such a work, would allow us to define heuristics for designing proactive fault tolerance.

REFERENCES

[1] O. Babaouglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, A. van Moorsel, and M. van Steen (Eds.) *Self-Star Properties in Complex Information Systems*. LNCS 3460. Springer-Verlag, 2005.

[2] *HATS Project*: Highly Adaptable and Trustworthy Software using formal models. *www.hats-project.eu/*.Accessed 20.03.2014

[3] P. Horn*,* Autonomic Computing*:* IBM's perspective on the State of Information Technology. http://researchweb.watson.ibm.com /autonomic/. Accessed 20.03.2014

[4] A .Iliasov, E. Troubitsyna, L. Laibinis, A.Romanovsky, and K.Varpaaniemi. Verfifying Mode Consistency for On-Board Satellite Softyware. In.Proc. SAFECOMP 2010, LNCS 6351, pp. 126-141, Springer, 2004.

[5] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K.Varpaaniemi, D. Ilic, T. Latvala, Developing Mode-Rich Satellite Software by Refinement in Event B . In: *Proc. of FMICS 2010,* LNCS 6371, pp. 50-66, Springer, 2010.

[6] L. Laibinis and E.Troubitsyna. Fault tolerance in a layered architecture: a general specification pattern in B. In Proc. of SEFM 2004. pp. 346-355, IEEE Computer Press, 2004.

[7] L. Laibinis, E. Troubitsyna, A. Iliasov and A. Romanovsky. Rigorous Development of Fault-Tolerant Agent Systems. *Rigorous Development of Complex Fault-Tolerant Systems*. LNCS 4157, pp. 241-260, Springer, 2006.

[8] J. C. Laprie, *Dependability: Basic Concepts and Terminology*. New York, Springer-Verlag, 1991.

[9] M. Salehie, L. Tahvildari: Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems* 4(2). ACM, 2009.

[10] F. Salfner, M. Lenk, and M. Malek: A survey of online failure prediction methods. *ACM Comput. Surv*. 42(3), 2010.

[11] K. Sere and E. Troubitsyna. Safety Analysis in Formal Specification. In *Proc. of FM'99,* LNCS 1709, pp. 1564 – 1583, Springer, 1999.

[12] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering Architectures from Running Systems. In *IEEE Transactions on Software Engineering*, Vol. 32(7), July 2006.

[13] A. Tarasyuk, I. Pereverzeva, E. Troubitsyna, T. Latvala, and L. Nummila, Formal Development and Assessment of a Reconfigurable On-Board Satellite System. In Proc. of *SAFECOMP 2012*, LNCS 7612, pp. 210–222, Springer-Verlag, 2012.

[14] E. Troubitsyna. Reliability assessment through probabilistic refinement. Nordic Journal of Computing 6(3), 320-342, 1999.

Paper VI.

E. Troubitsyna and K.Javed, “A Structured Approach to Architecting Fault Tolerant Services”, ICIW 2014, The Ninth International Conference on Internet and Web Applications and Services, pp. 99-104, July 2014, Paris, France.

A Structured Approach to Architecting Fault Tolerant Services

Elena Troubitsyna, Kashif Javed
Åbo Akademi University, Finland
Elena.Troubitsyna@abo.fi, Kashif.Javed@abo.fi

***Abstract*— Service-oriented computing offers an attractive paradigm to designing complex composite services by assembling readily-available services. The approach enables rapid service development and significantly increases productivity of the development. However, it also poses a significant challenge in ensuring quality of created services and in particular their fault tolerance. In this paper, we propose a systematic approach to architecting complex fault tolerant services. We demonstrate how to graphically model the architecture of composite services and augment it with various fault tolerance mechanisms. We propose an approach facilitating a systematic analysis of possible failures of the services, recovery actions and alternative solutions for achieving fault tolerance. Our approach supports structured guided reasoning about fault tolerance at different levels of abstraction. It allows the designers evaluate various architectural solutions at the design stage that helps to derive clean architectures and improve fault tolerance of developed complex services.**

Keywords - services; fault tolerance, architecture, service composition, service orchestration; failure modes and effect analysis

I. INTRODUCTION

Web-services [13] constitute one of the fastest growing areas of software engineering. With a strong support for compositionality, the process of developing an application essentially becomes a process of composing available services. Services – the basic building blocks of complex applications are platform and network independent components implementing computations that can be invoked by clients or other services.

To enable a rapid service composition, services define their properties in a standard and machine readable format. It enables service discovery, selection and binding. Service composition introduces the orchestration of the basic services to build applications. However, usually research on service orchestration focuses on defining the language for service composition that does not support reasoning about such essential features as fault tolerance. Such reasoning can be supported by dependability analysis and architectural modelling [5].

In this paper, we propose a systematic approach to architecting fault tolerant services. We demonstrate how to graphically model the architecture of composite services and augment it with various fault tolerance mechanisms. We propose static and dynamic solutions for introducing fault tolerance into the service composition. The structural solutions rely on availability of redundant service providers that can be requested to provide services in case of failures of the main service providers. This mechanism allows the designers to mask failures of the individual service providers. The dynamic solutions rely on re-execution of failed services to recover from the transient faults of services. This solution requires modifications of the service execution flow.

To facilitate design of complex fault tolerant services, in this paper, we introduce a systematic approach to analysing possible failure modes of services and defining fault tolerance measures. Our approach is inductive – it progressively analyses one component after another in the service execution flow, explores possible fault tolerance alternatives and systematically introduces them into the service architecture.

We believe that our approach supports structured guided reasoning about fault tolerance and enables efficient exploration of the design space. It allows the designers to evaluate various architectural solutions at the design stage that helps to derive clean architectures and improve fault tolerance of developed complex services.

The paper is structured as follows: in Section II, we demonstrate how to model a fault tolerant service from a service user's perspective. In Section III, we demonstrate how to unfold service architecture, i.e., explicitly represent the service composition and the service execution flow. We also propose different fault tolerance mechanisms that can be introduced to enhance fault tolerance. In Section IV, we introduce a structured approach to designing a fault tolerant architecture. Finally, in Section V, we overview the related work and discuss the presented work.

II. ABSTRACT MODELING OF FAULT-TOLERANT SERVICES

The main goal of introducing fault tolerance in the service architecture is to prevent a propagation of faults to the service interface level, i.e., to avoid a service failure [7] [9]. A fault manifests itself as *error* – an incorrect service

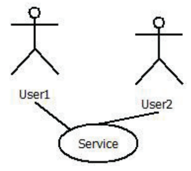


Figure1. Use case representation of a service.

state [9]. Once an error is detected, an error recovery should be initiated. Error recovery is an attempt to restore a fault-free state or at least to preclude system failure.

Error recovery aims at masking error occurrence or ensuring deterministic failure behaviour if the error cannot be masked. In the former case, upon detection of error, software executes certain actions to restore a fault-free system states and then guarantee normal service provisioning. In the latter case, the service provisioning is aborted and failure response is returned.

In this paper, we focus on the architectural graphical modelling [12] of fault tolerant services [13]. We demonstrate how to explicitly introduce handling of faulty behaviour into the service architecture. We follow the model-driven development paradigm and start our modelling from a high level of abstraction [8]. The consecutive model transformations introduce the detailed representation of the service architecture.

The high-level model of a fault tolerant service is given in Fig.1. The service is defined via its interactions with different service users. Each association connecting an external user and a service corresponds to a logical interface, as shown in Fig.2. The logical interfaces are attached to the class with ports. At the abstract modelling level, we treat a service as a black box with the defined logical interfaces.

The UML2 interfaces *I_ToService* and *I_FromService* define the request and request parameters of the service user. We formally describe the communication between a service and its user(s) in the *I_Communication* state machine as illustrated in Fig.3. The request *ser_req* received from the user is always replied: with the *ser_cnf* in case of success, with the *ser_fail_cnf* in case of unrecoverable failure and with the *ser_tfail_cnf* in case of a recoverable failure. Let us point out, that already at the abstract level of modelling, we explicitly introduce representation of faulty behaviour and reaction on it.

To exemplify an abstract modelling of a fault tolerant service, let us consider a *positioning service*. It provides the services for calculating the physical location of the service user.

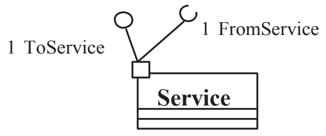


Figure 2. Abstract architectural diagram.

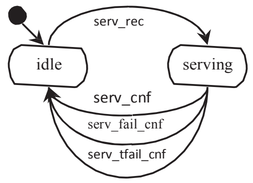


Figure 3. State diagram of communication.

As shown in Fig.4, the abstract model represents an interaction of the service with a user. An abstract architectural diagram defines an interface for communicating with the user. The state diagram formally defines the communication between the user and the service.

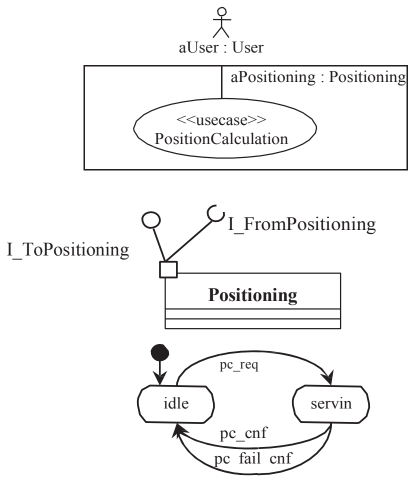


Fig.4. Modelling positioning service

The request to calculate the position is modelled by the event *pc_req*. In case of a normal execution, the positioning service returns the reply *pc_cnf*. Let us observe, that in our modelling we explicitly define the possibility of a service failure following the pattern proposed above. Indeed, in case of the unrecoverable failure, the positioning service returns *pc_fail_cnf*. In case of a recoverable failure, the service returns *pc_tfail_cnf*. Such a fault-tolerance explicit approach to modelling ensures that the service execution always terminates, i.e., the service never becomes unresponsive.

III. ARCHITECTURAL DECOMPOSITION

Our abstract modelling has defined the service from the service user's point of view. The model transformation presented next focuses on defining the composition that constitutes the overall service.

An execution of composite service consists of executing several subservices. Coordination of a service execution is performed by a *service manager* (sometimes called *service composer*). It is a dedicated software component that on the one hand, communicates with a service user and on the other hand, orchestrates the service execution flow.

To coordinate service execution, the service manager keeps the information about subservices and their execution order. It requests the corresponding service components to provide the required subservices and monitors the results of their execution.

Let us note, that any subservice might also be composed of several subservices, i.e., in its turn, the subservice execution might be orchestrated by its (sub)service manager. Hence, in general, a composite service might have several layers of hierarchy [5].

To model a composite service, we introduce the providers of the subservices into the abstract architectural service model. The model includes the external service

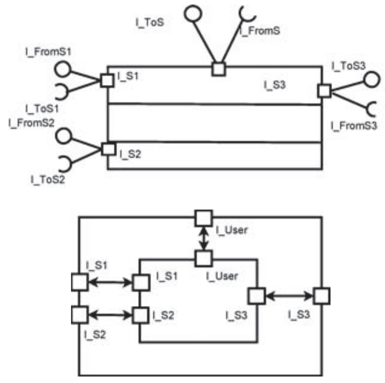


Figure 5. Architecture of a positioning service.

providers communicating with the aggregated service via their service director. For each association between the main service and the corresponding subservice, we define a logical interface. The logical interfaces are attached to the corresponding classes via the corresponding ports. This enables a structured representation of the modular structure of the composite service. The functional architecture is defined in terms of the service components, which encapsulate the functionality related to a single execution stage of another logical piece of functionality.

The architectural diagram of the position calculation [5] [14] – the composite service example described above is presented in Fig. 5. The service manager role is two-fold: it orchestrates service execution flow and handles communication with the service user. The dynamics of the execution flow is refined by introducing the corresponding sub-states in the service state as shown in Fig. 6.

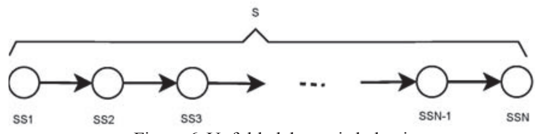


Figure 6. Unfolded dynamic behaviour.

Now, let us discuss the fault tolerant aspect of the composite services. Execution of any subservice can fail. To ensure fault tolerance of composite services, we propose a two-fold approach. On the one hand, we define a set of patterns [11] that allow us to introduce structural means for fault tolerance using various forms of redundancy. On the other hand, we propose to extend the responsibilities of a service manager, to implement dynamic error recovery. Next, we propose the architectural patterns for introducing structural fault tolerance and define the corresponding modeling artifacts.

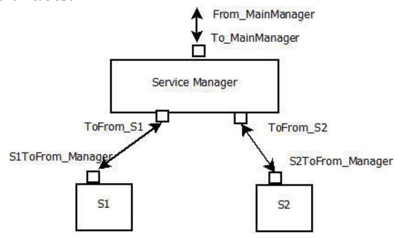


Figure 7. Duplication scheme.

Duplication pattern. The duplication is a simplest arrangement for structural fault tolerance. It can be introduced if there are two service components available that provide the same functionality. In this case, the services can be executed in parallel. A successful execution of a service by any out of two service components suffices for the successful service provisioning.

An architectural diagram of the duplication arrangement is given in Fig. 7. We introduce a dedicated service manager to take care of the execution of the duplicated service. The dynamical behavior of the duplication pattern is shown in Fig. 8. An alternative architectural approach would be to allow the main service manager to orchestrate this arrangement.

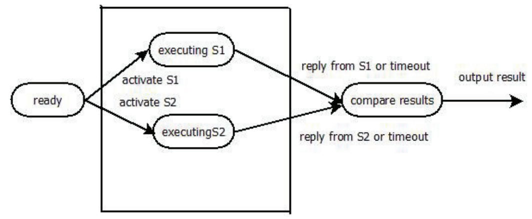


Figure 8. Dynamic behavior of duplication pattern.

Stand-by spare. This arrangement relies on availability of a spare service component implementing the desirable service. The spare is used only if the execution of the service by the main component fails. If the main service component succeeds in executing a service, the spare service component remains inactive. However, if the main service component fails to execute a service then the spare service component is requested to provide the service.

The stand-by spare arrangement can be implemented with and without an introduction of the dedicated service director. The design decision depends on the complexity of the composite service, i.e., whether the design of the main service manager would become too complex with the introduction of this additional responsibility.

The architecture of the stand-by-spare implemented with the dedicated service manager coincides with the duplication pattern. However, the dynamic behavior is different as shown in Fig.9.

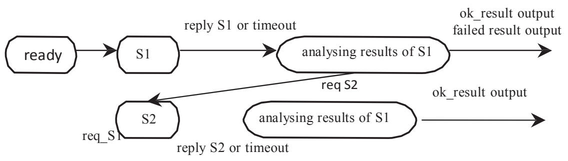


Figure 9. Dynamic behavior of stand-by spare.

Triple modular redundancy pattern. A more complicated scheme for structural redundancy – triple modular redundancy is shown in Fig.10. The precondition for implementing it is that we have three service components available that provide identical services with the same functionality. All three service components receive the same service request and work in parallel. The results of the service execution are sent to a voting element.

The voting element is a dedicated software component that performs comparison of the results and produces the final result. The voting element takes a majority view over the produced results of the successfully executed services and outputs it as the final result of the service execution.

In the context of the service-oriented computing, the voting component might be implemented in two different ways: it might output the results after receiving the first two replies or it might start to act only after the certain deadline when all non-failed services have replied.

Let us discuss a difference between triple modular redundancy scheme adopted in hardware and services. In hardware context, the scheme can mask failure of a single component by adopting the majority view. In the service-oriented context, it gives more fault tolerance options. Indeed, if two out of three services failed to reply within the timeout, the voter component can be design to simply output the result of the non-failed service. Obviously, in case of a failure of a single service, it gives better fault tolerance guarantees, because it can compare the results of two non-failed services and take the one, which is more accurate as the output.

Since the triple modular redundancy scheme has a rather complex architecture by itself, we propose to introduce a dedicated service manager to integrate the arrangement in the architecture of a composite service. The proposal is depicted in Fig. 10.

The dynamic behavior of the triple modular arrangement is depicted in Fig.11. Here, the dedicated service manager performs voting before outputting the service result.

The static redundancy schemes require availability of redundant service components and hence, sometimes,

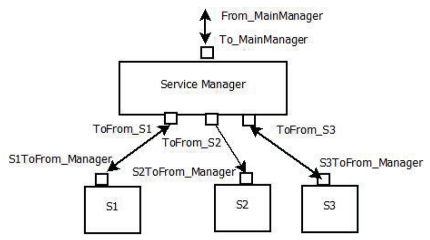


Figure 10. Architecture of triple modular redundancy.

might be non-implementable. However, they provide an efficient means to cope with permanent service failure. In contrast, dynamic fault tolerance relies on service re-execu-

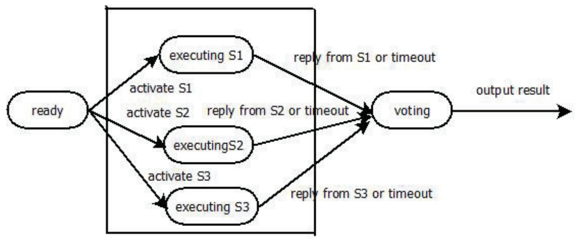


Figure 11. Dynamic behaviour of triple modular redundancy.

tion to increase the chances of the successful service execution and does not require an availability of the redundant service components. Obviously, the dynamic fault tolerance solutions can cope with transient failures.

To leverage fault tolerance of a composite service, the service manager might alter the normal flow of service execution to dynamically cope with failures. For instance, it might repeat service execution, roll-back or abort service execution.

If service execution failed, but the returned exception indicates that the error is transient then by re-executing the failed subservice, the service manager might recover from the error. The service execution flow is shown in Fig.12.

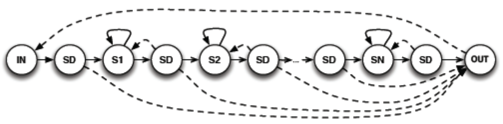


Figure 12. Service execution flow.

If service execution failed but the returned exception indicates that the error is unrecoverable and there are no alternative services available, then the service manager can abort the entire service execution and return failure response.

Obviously, designing fault tolerant composite services is a non-trivial task that requires a systematic support. In the next section, we propose an approach to systematic development of fault tolerant architecture by a structured analysis of failure modes of the services and fault tolerance schemes.

IV. DEVELOPMENT OF A FAULT TOLERANT SERVICE ARCHITECTURE

The main motivation behind our approach is to facilitate a structured disciplined derivation of fault tolerant service architecture. Essentially, we define the guidelines for analyzing faulty behavior of the services and deciding on the mechanisms for fault tolerance.

Our approach is inspired by the Failure Modes and Effect Analysis (FMEA) technique. FMEA [16] is an inductive analysis method, which allows designers to systematically study the causes of components faults, their effects and means to cope with these faults. FMEA is used to assess the effects of each failure mode of a component on the various functions of the system as well as to identify the failure modes significantly affecting dependability of the system.

FMEA step-by-step selects the individual components of the system, identifies possible causes of each failure mode, assesses consequences and suggests remedial actions. The results of FMEA are usually represented in the tabular form that contains the following fields: component name, failure mode, possible cause, local effect, system effect, detection, and remedial action.

Let us exemplify the proposed approach. Assume that a service *S1* is a part of the composite service *S*. The services *S11* and *S12* have identical functionality. Assume that the service *S1* might experience transient silent failures, i.e., become temporally irresponsive. Such failures can be detected by timeout. Then we can arrange services into a triple modular redundancy scheme. The structured analysis of the fault tolerance arrangement around the service *S1* according to the proposed approach is shown in Table I.

TABLE I. TRANSIENT FAILURE ANALYSIS

Service	S1
Failure mode	Transient silent failure
Detection	Timeout
Available redundancy	S11, S12
Structural redundancy	Triple modular redundancy arrangement. Result is produced upon timeout
Recovery	Masking failure by use of triple modular redundancy arrangement. In case of simultaneous failure of S1, S11 and S12 repeat execution

Let us now assume that a service *S2* is also part of the composite service *S*. Assume that the service *S2* might experience transient failures that are identified by receiving the response *S2_tfail_cnf* from it. Since no redundant service components are available for this case and the service failure is detectable with the corresponding notification, we can rely on dynamic redundancy to cope with failures of S2. The structured analysis of the fault tolerance arrangement around the service *S2* according to the proposed approach presented in Table II.

TABLE II. FAILURE MODE ANALYSIS

Service	S2
Failure mode	Transient detectable failure
Detection	*S2_tfail_cnf* response
Available redundancy	No
Structural redundancy	No
Recovery	Re-execute service. Maximal allowed number of retries is 3.

It easy to observe that reliance on the proposed approach facilitates structured derivation of fault tolerance architecture for both structured and dynamic fault tolerance schemes.

As a result of introducing various means for fault tolerance, we also should modify the design of the service manager. Fig 13 depicts the modified flow with a retry.

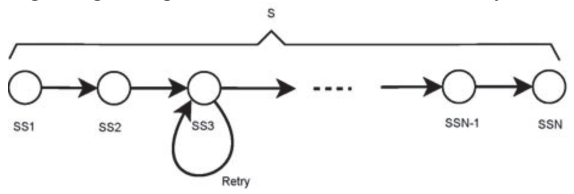


Figure 13. Execution flow with retry.

The process of introducing fault tolerance mechanisms can be iteratively applied to unfold all the architectural layers. As a result of this process, we obtain a hierarchical structure of service managers augmented with fault tolerance properties.

V. RELATED WORK AND CONCLUSIONS

While the topic of service orchestration and composition has received significant research attention, the fault tolerance aspect is not so well addressed. Liang [10] proposes a fault-tolerant web service on SOAP (called FT-SOAP) using the service approach. It extends the standard WSDL by proposing a new element to describe the replicated web services. The client side SOAP engine searches for the next available backup from the group WSDL and redirects the request to the replica if the primary server failed. It is a rather complex mechanism that hinders interoperability.

Artix [2] is IONA's Web services integration product. It provides a WSDL-based naming service by Artix Locator. Multiple instances of the same service can be registered under the same name with an Artix Locator.

When service consumers request a service, the Artix Locator selects the service instance based on a load-balancing algorithm from the pool of service instances. It provides useable services for the service consumers. An active UDDI mechanism [4] enables an extension of UDDI's invocation API to enable fault-tolerant and dynamic service invocation. Its function is similar to the Artix Locator. A dependable Web services framework is proposed in [1]. Once a failure for one specific service occurs, the proxy raises a "WebServiceNotFound" exception and downloads its handler from DeW. The exception handling chooses another location that hosts the same service and re-invoks the method automatically. The main goal of DeW is to realize physical-location-independence. Providing fault-tolerance capability for composite Web service has also been discussed in [3].

A formal approach [15] [17] to introducing fault tolerance to the service architecture has been proposed in [5] [6]. This work extends the set of architectural patterns that can be introduced to achieve fault tolerance as well as propose a systematic support for deriving fault tolerance solutions.

In this paper, we have proposed a systematic approach to architecting fault tolerant services. We demonstrated how to graphically model the architecture of composite services and augment it with various fault tolerance mechanisms. We defined a set of static and dynamic solutions for introducing fault tolerance into the service composition. The proposed mechanisms can cope with different types of failures to increase reliability of complex composite services.

To facilitate design of fault tolerance mechanisms, we proposed an approach to a structured analysis of possible failure modes of services and introducing fault tolerance measures. The proposed approach is inductive – it progressively analyses services in the execution flow, explores possible fault tolerance alternatives and systematically introduces them into the service architecture.

We believe that our approach supports structured guided reasoning about fault tolerance and enables efficient exploration of the design space while architecting complex composite services.

ACKNOWLEDGMENT

Troubitsyna thanks the Need for Speed program. http://www.digile.fi/N4S for a financial support.

REFERENCES

[1] E. Alwagait, S. Ghandeharizadeh, "A Dependable Web Services Framework" 14th International Workshop on Research Issues on Data Engineering 2004. [Online]. Available from http://fac.ksu.edu.sa/alwagait/publication/31143 2014.30.05.

[2] Artix Technical Brief. [Online]. Available from http://www.iona.com/artix 2014.30.05.

[3] V. Dialani, S. Miles, L.Moreau, D. Roure, M. Dialani, "Transparent fault tolerance for web services based architectures". 8th Europar Conference (EULRO-PAR02), Springer 2002, pp. 889-898. ISBN:3-540-44049-6.

[4] M. Jeckle, B. Zengler, "Active UDDI-An Extension to UDDI for Dynamic and Fault Tolerant Service Invocation" 2nd International Workshop on Web and Databases, Springer 2002, pp. 91-99. ISBN:3-540-00745-8.

[5] L. Laibinis, E. Troubitsyna and S. Leppänen, "Service-Oriented Development of Fault Tolerant Communicating Systems: Refinement Approach" International Journal on Embedded and Real-Time Communication Systems, vol. 1, pp. 61-85, Oct. 2010, DOI: 10.4018/jertcs.2010040104.

[6] L. Laibinis, E. Troubitsyna, A. Iliasov, A. Romanovsky, "Rigorous development of fault-tolerant agent systems", In in M. Butler, C. Jones, A. Romanovsky and E. Troubitsyna (Eds.), Rigorous Development of Complex Fault-Tolerant Systems, LNCS 4157, pp. 241-260, Springer 2006, ISBN 978-3-642-00867-2.

[7] L. Laibinis, E. Troubitsyna, S. Leppänen, J. Lilius and Q. Malik, "Formal Service-Oriented Development of Fault Tolerant Communicating Systems", in M. Butler, C. Jones, A. Romanovsky and E. Troubitsyna (Eds.), Rigorous Development of Complex Fault-Tolerant Systems, LNCS 4157, pp. 261-287, Springer 2006, ISBN 978-3-642-00867-2.

[8] L. Laibinis, E. Troubitsyna "Fault Tolerance in use-case modelling", In Workshop on Requirements for High Assurance Systems (RHAS 05), [Online]. Available from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.8 4.4950 2014.01.05.

[9] J. C. Laprie. Dependability: Basic Concepts and Terminology. Springer-Verlag, 1991.

[10] D. Liang, C. L. Fang, C. Chen, F. X, Lin. "Fault-tolerant web service". Tenth Asia-Pacific Software Engineering Conference, IEEE Press, Dec. 2003, pp.56-61, ISBN 973-4-642-01867-1.

[11] I. Lopatkin, A. Iliasov, A. Romanovsky, Y. Prokhorova, E. Troubitsyna, "Patterns for representing FMEA in formal specification of control systems" High-Assurance Systems Engineering Conference (HASE), IEEE Nov 2011, pp. 146 – 151, ISBN 978-1-4673-0107-7.

[12] J. Rumbaugh, I. Jakobson, and G .Booch, The Unified Modelling Language Reference Manual. Addison-Wesley, 1998.

[13] Web Services Architecture Requirements. [Online] Available from http://www.w3.org/TR/wsareqs. 2014.01.05.

[14] 3GPP. Technical specification 25.305: Stage 2 functional specification of UE positioning in UTRAN. Available at http://www.3gpp.org/ftp/Specs/html-info/25305.htm. Accessed 01.05.2014.

[15] K. Sere, E. Troubitsyna, "Safety analysis in formal specification" In Formal Methods (FM'1999), Springer Sep. 1999, pp 1564-1583, ISBN:3-540-66588-9.

[16] N. Storey. Safety-critical computer systems. Addison-Wesley, 1996.

[17] E. Troubitsyna. "Elicitation and specification of safety requirements". In Third International Conference on Systems (ICONS 08), IEEE Apr. 2008, pp. 202-207, ISBN978-0-7695-3105-2.

ISBN 978-952-12-3553-5
ISSN 1239-1883

Kashif Javed

Model-Driven Development and Verification of Fault Tolerant Systems