Architecting Fault-Tolerant Software Systems

Hasan Sözer

Architecting Fault-Tolerant Software Systems

Hasan Sözer

CTIT Ph.D. thesis series no. 09-135. Centre for Telematics and Information Technology (CTIT), P.O. Box 217 - 7500 AE Enschede, The Netherlands.

This work has been carried out as part of the Trader project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Dutch Government under the Bsik program. The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

ISBN 978-90-365-2788-0
ISSN 1381-36-17 (CTIT Ph.D. thesis series no. 09-135)
IPA Dissertation Series 2009-05

Cover design by Hasan Sözer
Printed by PrintPartners Ipskamp, Enschede, The Netherlands

Architecting Fault-Tolerant Software Systems

DISSERTATION

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus,
Prof. Dr. H. Brinksma,
on account of the decision of the graduation committee,
to be publicly defended
on Thursday the 29th of January 2009 at 16.45

by

Hasan Sözer

born on the 21st of August 1980
in Bursa, Turkey

This dissertation is approved by

Prof. Dr. Ir. M. Akşit (promoter)
Dr. Ir. Bedir Tekinerdoğan (assistant promoter)

"The more you know, the more you realize you know nothing."

- Socrates

Acknowledgements

When I was a M.Sc. student at Bilkent University, I have met with Bedir Tekinerdoğan. He was a visiting assistant professor there at that time. Towards the end of my M.Sc. studies, he has notified me about the vacancy for a Ph.D. position at the University of Twente. He has also recommended me for this position. First of all, I would like to thank him for the faith he had in me. Following my admission to this position, he became my daily supervisor and we have been working very closely thereafter. I have always been impressed by his ability to abstract away key points out of details and his writing/presentation skills based on a true empathy towards the intended audience. I would like to thank him for his contributions to my intellectual growth and for his continuous encouragement, which has been an important source of motivation for me.

I have carried out my Ph.D. studies at the software engineering group lead by Mehmet Akşit. We have had regular meetings with him to discuss my progress and future research directions. In these meetings, I have sometimes been exposed to challenging critics but always with a positive, optimistic attitude and encouragement. Over the years, I have witnessed his ability to foresee pitfalls and I have been convinced about the accuracy of his predictions in research. I would like to thank him for his reliable guidance.

During my studies, I have also had the opportunity to work together with Hichem Boudali and Mariëlle Stoelinga from the formal methods group. I have learned a lot from them and an important part of this thesis (Section 5.10) presents the results of our collaboration. I would like to thank them for their contribution.

I would like to thank to the members of my Ph.D. committee: Jan Broenink, Arjan van Gemund, Rogério de Lemos, Alexander Romanovsky, and Gerard Smit for spending their valuable time and energy to evaluate my work. Their useful comments enabled me to dramatically improve this thesis.

I would like to thank to the members of the Trader project for their useful feedback during our regular project meetings. In particular, David Watts, Jozef Hooman and Teun Hendriks have reviewed my work closely. Ben Pronk brought up the research direction on local recovery, which later happened to be the main focus of my work. He has also spent his valuable time to provide us TV domain knowledge together with Rob Golsteijn. Previously we had several discussions with Iulian Nitescu, Paul L. Janson and Pierre van de Laar on failure scenarios, fault/error/failure classes and recovery strategies. These discussions have also directly or indirectly contributed to this thesis.

The members of the software engineering group have provided me useful feedback during our regular seminars. I would like to thank them also for the comfortable working environment I have had. In particular, I thank my roommates over the years: Joost, Christian and Somayeh. In addition to the Dutch courses provided by the university, Joost has given me a ‘complementary’ course on Dutch language and Dutch culture. He has also read and corrected my official Dutch letters, which would have caused quite some trouble if they were not corrected. Christian and Somayeh have always been open to give their opinion about any issue I may bring up and help me if necessary.

I would like to thank Ellen Roberts-Tieke, Joke Lammerink, Elvira Dijkhuis, Hilda Ferweda and Nathalie van Zetten for their invaluable administrative support.

To be able to finish this work, first of all I had to feel secure and comfortable in my social environment. In the following, I would like to extend my gratitude to people, who have provided me such an environment during the last four years.

When I first arrived in Enschede, Gürcan was one of the few people I knew at the university. He has helped me a lot to get acquainted with the new environment. He has provided me a useful set of survival strategies to deal with never-ending official procedures. The set of strategies has been later extended for surviving at mountains and at the military service as well.

I have been sharing an apartment with Espen during the last three years. The life is a lot easier if you always have a reliable friend around to talk to. Espen is very effective in killing stress and boosting courage in any circumstance (almost like alcohol, but almost healthy at the same time). Besides Espen, I had the chance to meet with several other good friends while I was living at a student house (the infamous 399) in the campus. I am sure that we will keep in touch in the future, in one way or another.

Although I have been living abroad, I have also had the chance to meet with many new Turkish friends during my studies. They have became very close friends of mine and they have helped me not to feel so much homesick. There are many people to count in this group and I will shortly refer to them as ‘öztwenteliler’. Selim and Emre are also in this group and I especially thank them for “supporting” me during my defense.

I would like to thank all the people who have contributed to Tusat. Similarly, I am grateful to people who have volunteered to work for making our life more social and enjoyable in the university, for example, members of Esn Twente over the years. Stichting Kleurrijke Dans has also made my life more colorful lately. In addition, I would like to thank all my friends who accompany me in recreational trips and various other activities during the last four years.

I have had endless love and support from my family throughout my life. I thank foremost my parents for always standing by me regardless of the geographic distance between us.

Abstract

The increasing size and complexity of software systems makes it hard to prevent or remove all possible faults. Faults that remain in the system can eventually lead to a system failure. Fault tolerance techniques are introduced for enabling systems to recover and continue operation when they are subject to faults. Many fault tolerance techniques are available but incorporating them in a system is not always trivial. We consider the following problems in designing a fault-tolerant system. First, existing reliability analysis techniques generally do not prioritize potential failures from the end-user perspective and accordingly do not identify sensitivity points of a system. Second, existing architecture styles are not well-suited for specifying, communicating and analyzing design decisions that are particularly related to the fault-tolerant aspects of a system. Third, there are no adequate analysis techniques that evaluate the impact of fault tolerance techniques on the functional decomposition of software architecture. Fourth, realizing a fault-tolerant design usually requires a substantial development and maintenance effort.

To tackle the first problem, we propose a scenario-based software architecture reliability analysis method, called SARAH that benefits from mature reliability engineering techniques (i.e. FMEA, FTA) to provide an early reliability analysis of the software architecture design. SARAH evaluates potential failures from the end-user perspective to identify sensitive points of a system without requiring an implementation.

As a new architectural style, we introduce Recovery Style for specifying fault-tolerant aspects of software architecture. Recovery Style is used for communicating and analyzing architectural design decisions and for supporting detailed design with respect to recovery.

As a solution for the third problem, we propose a systematic method for optimizing the decomposition of software architecture for local recovery, which is an effective fault tolerance technique to attain high system availability. To support the method, we have developed an integrated set of tools that employ optimization techniques, state-based analytical models (i.e. CTMCs) and dynamic analysis on the system.

The method enables the following: *i*) modeling the design space of the possible decomposition alternatives, *ii*) reducing the design space with respect to domain and stakeholder constraints and *iii*) making the desired trade-off between availability and performance metrics.

To reduce the development and maintenance effort, we propose a framework, FLORA that supports the decomposition and implementation of software architecture for local recovery. The framework provides reusable abstractions for defining recoverable units and for incorporating the necessary coordination and communication protocols for recovery.

Contents

Chapter 1

Introduction

A system is said to be *reliable* [3] if it can continue to provide the correct service, which implements the required system function. A *failure* occurs when the delivered service deviates from the correct service. The system state that leads to a failure is defined as an *error* and the cause of an error is called a *fault* [3]. It becomes harder to prevent or remove all possible faults in a system as the size and complexity of software increases. Moreover, the behavior of current systems are affected by an increasing number of external factors since they are generally integrated in networked environments, interacting with many systems and users. It may therefore not be economically and/or technically feasible to implement a fault-free system. As a consequence, the system needs to be able to tolerate faults to increase its *reliability*. Potential failures can be prevented by designing the system to be recoverable from errors regardless of the faults that cause them. This is the motivation for *fault-tolerant design*, which aims at enabling a system to continue operation in case of an error. By this way, the system can remain *available* to its users, possibly with reduced functionality and performance rather than failing completely. In this thesis, we introduce methods and tools for the application of fault tolerance techniques to increase the reliability and availability of software systems.

1.1 Thesis Scope

The work presented in this thesis has been carried out as a part of the TRADER[1] [120] project. The objective of the project is to develop methods and tools for ensuring reliability of digital television (DTV) sets. A number of important trends can be observed in the development of embedded systems like DTVs. First, due to the high industrial competition and the advances in hardware and software technology, there is a continuous demand for products with more functionality. Second, the implementation of functionality is shifting from hardware to software. Third, products are not solely developed by just one manufacturer only but it is host to multiple parties. Finally, embedded systems are more and more integrated in networked environments that affect these systems in ways that might not have been foreseen during their construction. Altogether, these trends increase the size and complexity of software in embedded systems and as such make software faults a primary threat for reliability.

For a long period, reliability and fault-tolerant aspects of embedded systems have been basically addressed at hardware level or source code. However, in face of the current trends it has now been recognized that reliability analysis should focus more on software components. In addition, incorporation of some fault tolerance techniques should be considered at a higher abstraction level than source code. It must be ensured that the design of the ***software architecture*** supports the application of necessary fault tolerance techniques. Software architecture represents the gross-level structure of the system that directly influences the subsequent analysis, design and implementation. Hence, it is important to evaluate the impact of fault tolerance techniques on the software architecture. By this way, the quality of the system can be assessed before realizing the fault-tolerant design. This is essential to identify potential risks and avoid costly redesigns and reimplementations.

Different type of fault tolerance techniques are employed in different application domains depending on their requirements. For *safety-critical systems*, such as the ones used in nuclear power plants and airplanes, safety is the primary concern and any failure that can cause harm to people and environment must be prevented. In that context, the additional cost of the fault-tolerant design due to the required hardware/software resources is a minor issue. In the TRADER project [120], we have focused on the consumer electronics domain, in particular, DTV systems. For such systems, which are probably less subjected to catastrophic failures, the ***cost*** and the ***perception of the user*** turn out to be the primary concerns, instead. These systems are very cost-sensitive and failures that are not directly perceived by the user can be accepted to some extent, whereas failures that can be directly

[1]TRADER stands for Television Related Architecture and Design to Enhance Reliability.

observed by the user require a special attention. In this context, the fault tolerance techniques to be applied must be evaluated with respect to their additional costs and their effectiveness based on the user perception.

1.2 Motivation

Traditional software fault tolerance techniques are mostly based on design diversity and replication because software failures are generally caused by design and coding faults, which are permanent [58]. So, the erroneous system state that is caused by such faults has also been assumed to be permanent. On the other hand, software systems are also exposed to so-called *transient faults*. These faults are mostly activated by timing issues and peak conditions in workload that could not have been anticipated before. Errors that are caused by such faults are likely to be resolved when the software is re-executed after a clean-up and initialization [58]. As a result, it is possible to design a system that can recover from a significant fraction of errors [16] without replication and design diversity and as such without requiring substantial hardware/software resources [58]. Many such fault tolerance techniques are available but developing a fault-tolerant system is not always trivial.

First of all, we need to know which fault tolerance techniques to select and where in the system to apply the selected set of techniques. Due to the cost-sensitivity of consumer electronics products like DTVs, it is not feasible to design a system that can tolerate all the faults of each of its elements. Thus, we need to analyze potential failures and prioritize them based on user perception. Accordingly, we should identify sensitive elements of the system, whose failures might cause the most critical system failures. The set of fault tolerance techniques should be then selected based on the type of faults activated by the identified sensitive elements. Existing reliability analysis techniques do not generally prioritize potential failures from the end-user perspective and they do not identify sensitive points of a system accordingly.

After we analyze the system and select the set of fault tolerance techniques accordingly, we should adapt the software architecture description to specify the fault-tolerant design. The software architecture of a system is usually described using more than one architectural view. Each view supports the modeling, understanding, communication and analysis of the software architecture for different concerns. This is because current software systems are too complex to represent all the concerns in one model. An analysis of the current practice for representing architectural views reveals that they focus mainly on functional concerns and are not well-suited for communicating and analyzing design decisions that are particularly related to the fault-tolerant aspects of a system. New architectural styles are required to be able to document the software architecture from the fault tolerance point of view.

Fault tolerance techniques may influence the decomposition of software architecture. *Local recovery* is such a fault tolerance technique, which aims at making the system ready for correct service as much as possible, and as such attaining high system *availability* [3]. For achieving local recovery the architecture needs to be decomposed into separate units (i.e. *recoverable units*) that can be recovered in isolation. Usually there are many different alternative ways to decompose the system for local recovery. Increasing the number of recoverable units can provide higher availability. However, this will also introduce an additional performance overhead since more modules will be isolated from each other. On the other hand, keeping the modules together in one recoverable unit will increase the performance, but will result in a lower availability since the failure of one module will affect the others as well. As a result, for selecting a decomposition alternative we have to cope with a trade-off between availability and performance. There are no adequate integrated set of analysis techniques to directly support this trade-off analysis, which requires optimization techniques, construction and analysis of quality models, and analysis of the existing code base to automatically derive dependencies between modules of the system. We need the utilization and integration of several analysis techniques to optimize the decomposition of software architecture for recovery.

The optimal decomposition for recovery is usually not aligned with the existing decomposition of the system. As a result, the realization of local recovery, while preserving the existing decomposition, is not trivial and requires a substantial development and maintenance effort [26]. Developers need to be supported for the implementation of the selected recovery design.

Accordingly, this thesis provides software architecture modeling, analysis and realization techniques to improve the reliability and availability of software systems by introducing fault tolerance techniques.

1.3 The Approach

In the following subsections, we summarize the approaches that we have taken for supporting the design and implementation of fault-tolerant software systems. The overall goal is to employ fault tolerance techniques that can mainly tolerate transient faults and as such to improve the reliability and availability of cost-sensitive systems from the user point of view.

1.3.1 Software architecture reliability analysis using failure scenarios

Our first approach aims at analyzing the potential failures and the sensitivity points at the software architecture design phase before the fault-tolerant design is implemented. Since implementing the software architecture is a costly process, it is important to predict the quality of the system and identify potential risks, before committing enormous organizational resources [31]. Similarly, it is of importance to analyze the hazards that can lead to failures and to analyze their impact on the reliability of the system before we select and implement fault tolerance techniques. For this purpose, we introduce a software architecture reliability analysis approach (SARAH) that benefits from mature reliability engineering techniques and scenario-based software architecture analysis to provide an early software reliability analysis. SARAH defines the notion of failure scenario model that is based on the Failure Modes and Effects Analysis method (FMEA) in the reliability engineering domain. The failure scenario model is applied to represent so-called failure scenarios that define a Fault Tree Set (FTS). FTS is used for providing a severity analysis for the overall software architecture and the individual architectural elements. Despite conventional reliability analysis techniques which prioritize failures based on criteria such as safety concerns, in SARAH failure scenarios are prioritized based on severity from the end-user perspective. The analysis results can be used for identifying so-called *architectural tactics* [5] to improve the reliability. Hereby, architectural tactics form building blocks of design patterns for fault tolerance.

1.3.2 Architectural style for recovery

Once we have selected the appropriate fault tolerance techniques and related architectural tactics, they should be incorporated into the existing software architecture. Introduction of fault tolerance mechanisms usually requires dedicated architectural elements and relations that impact the software architecture decomposition. Our second approach aims at modeling the resulting decomposition explicitly by providing a practical and easy-to-use method to document the software architecture from a recovery point of view. For this purpose, we introduce the *recovery style* for modeling the structure of the system related to the recovery concern. It is used for communicating and analyzing architectural design decisions and supporting detailed design with respect to recovery. The recovery style considers *recoverable units* as first class architectural elements, which represent the units of isolation, error containment and recovery control. The style defines basic relations for coordination and application of recovery actions. As a further specialization of the recovery style, the *local recovery style* is provided, which is used for documenting a local recovery design including the decomposition of software architecture into recoverable units and the way that these units are controlled.

1.3.3 Quantitative analysis and optimization of software architecture decomposition for recovery

To introduce local recovery to the system, first we need to select a decomposition among many alternatives. We propose a systematic approach dedicated to optimizing the decomposition of software architecture for local recovery. To support the approach, we have developed an integrated set of tools that employ *i*) dynamic program analysis to estimate the performance overhead introduced by different decomposition alternatives, *ii*) state-based analytical models (i.e. CTMCs) to estimate the availability achieved by different decomposition alternatives, and *iii*) optimization techniques for automatic evaluation of decomposition alternatives with respect to performance and availability metrics. The approach enables the following.

- modeling the design space of the possible decomposition alternatives
- reducing the design space with respect to domain and stakeholder constraints
- making the desired trade-off between availability and performance metrics

With this approach, the designer can systematically evaluate and compare decomposition alternatives, and select an optimal decomposition.

1.3.4 Framework for the realization of software architecture recovery design

After the optimal decomposition for recovery is selected, the software architecture should be partitioned accordingly. In addition, new supplementary architectural elements and relations should be implemented to enable local recovery. To reduce the resulting development and maintenance efforts we introduce a framework, FLORA that supports the decomposition and implementation of software architecture for local recovery. The framework provides reusable abstractions for defining recoverable units and the necessary coordination and communication protocols for recovery. Using our framework, we have introduced local recovery to the open-source media player called MPlayer for several decomposition alternatives. We have then performed measurements on these implementations to validate the results of our analysis approaches.

1.4 Thesis Overview

The thesis is organized as follows.

Chapter 2 provides background information and a set of definitions that is used throughout this thesis. It introduces the basic concepts of reliability, fault tolerance and software architectures.

Chapter 3 presents the software architecture reliability analysis method (SARAH). SARAH is a scenario-based analysis method, which aims at providing an early evaluation and feedback at the architecture design phase. It utilizes mature reliability engineering techniques to prioritize failure scenarios from the user perspective and identifying sensitive elements of the architecture accordingly. The output of SARAH can be utilized as an input by the techniques that are introduced in Chapter 4 and Chapter 5. This chapter is a revised version of the work described in [111], [117], and [118].

Chapter 4 introduces a new architectural style, called *Recovery Style* for modeling the structure of the software architecture that is related to the fault tolerance properties of a system. The style is used for communicating and analyzing architectural design decisions and supporting detailed design with respect to recovery. This style is used in Chapter 6 to represent the designs to be realized. It also supports the understanding of Chapter 5. This chapter is a revised version of the work described in [110].

Chapter 5 proposes a systematic approach dedicated to optimizing the decomposition of software architecture for local recovery. In this chapter, we explain several analysis techniques and tools that employ dynamic analysis, analytical models and optimization techniques. These are all integrated to support the approach.

Chapter 6 presents the framework FLORA that supports the decomposition and implementation of software architecture for local recovery. The framework provides reusable abstractions for defining recoverable units and the necessary coordination and communication protocols for recovery. This chapter is a revised and extended version of the work described in [112].

Chapter 7 provides our conclusions. The evaluations, discussions and related work for the particular contributions are provided in the corresponding chapters.

An overview of the main chapters is depicted in Figure 1.1. Hereby, the rounded rectangles represent the chapters of the thesis. The solid arrows represent the recommended reading order of these chapters. After reading Chapter 2, the reader can immediately start reading Chapter 3, 4 or 5. Chapter 4 should be read before Chapter 6. All the other chapters are self-contained. All the chapters provide complementary work and the works that are presented through chapters 4 to 6 are directly related.

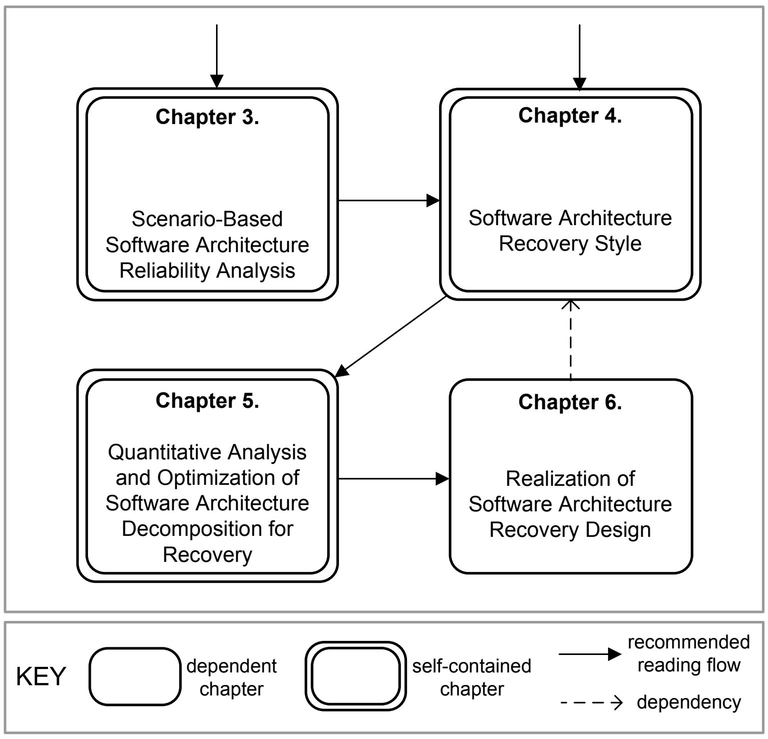

Chapter 3.
Scenario-Based Software Architecture Reliability Analysis
Chapter 4.
Software Architecture Recovery Style
Chapter 5.
Quantitative Analysis and Optimization of Software Architecture Decomposition for Recovery
Chapter 6.
Realization of Software Architecture Recovery Design
KEY
dependent chapter
self-contained chapter
recommended reading flow
dependency

Figure 1.1: The main chapters of the thesis and the recommended reading order.

Chapter 2

Background and Definitions

In our work, we utilize concepts and techniques from both the areas of dependability and software architectures. In this chapter, we provide background information on these two areas and we introduce a set of definitions that will be used throughout the thesis.

2.1 Dependability and Fault Tolerance

Dependability is the ability of a system to deliver service that can justifiably be trusted [3]. It is an integrative concept that encompasses several quality attributes including reliability, availability, safety, integrity and maintainability. A system is considered to be dependable if it can avoid failures (service failures) that are more frequent or more severe than is acceptable [3]. A *failure* occurs when the delivered service of a system deviates from the required system function [3]. An *error* is defined as the system state that is liable to lead to a failure and the cause of an error is called a *fault* [3]. Figure 2.1 depicts the fundamental chain of these concepts that leads to a failure. As an example, assume that a software developer allocates an insufficient amount of memory for an input buffer. This is the fault. At some point during the execution of the software, the size of the incoming data overflows this buffer. This is the error. As a result, the operating system kills the corresponding process and the user observes that the software crashes. This is the failure.

$$\begin{tikzcd}\text{Fault} \arrow[r, "\text{activation}"] & \text{Error} \arrow[r, "\text{propagation}"] & \text{Failure}\end{tikzcd}$$

Figure 2.1: The fundamental chain of dependability threats leading to a failure

Figure 2.1 shows the simplest possible chain of dependability threats. Usually, there are multiple errors involved in the chain, where an error propagates to other errors and finally leads to a failure [3].

2.1.1 Dependability and Related Quality Attributes

Dependability encompasses the following quality attributes [3]:

- **reliability:** continuity of correct service.
- **availability:** readiness for correct service.
- **safety:** absence of catastrophic consequences on the user(s) and the environment.
- **integrity:** absence of improper system alterations.
- **maintainability:** ability to undergo modifications and repairs.

Depending on the application domain, different emphasis might be put on different attributes. In this thesis, we have considered *reliability* and *availability*, whereas safety, integrity and maintainability are out of the scope of our work. Reliability and availability are very important quality attributes in the context of fault-tolerant systems and they are closely related. Reliability is the ability of a system to perform its required functions under stated conditions for a specified period of time. That is the ability of a system to function without a failure. Availability is the proportion of time, where a system is in a functioning condition. Ideally, a fault-tolerant system can recover from errors before any failure is observed by the user (i.e. reliability). However, this is practically not always possible and the system can be unavailable during its recovery. After a fault is activated, a fault-tolerant system must become operational again as soon as possible to increase its availability.

Even if there are no faults activated, fault tolerance techniques introduce a performance overhead during the operational time of the system. The overhead can be caused, for instance, by monitoring of the system for error detection, collecting and logging system traces for diagnosis, saving data for recovery and wrapping system elements for isolation. For this reason, in addition to reliability and availability, we consider *performance* as a relevant quality attribute in this work although it is not a dependability attribute. Performance is defined as the degree to which a system accomplishes its designated functions within given constraints, such as speed and accuracy [60].

In traditional reliability and availability analysis, a system is assumed to be either up and running, or it is not. However, some fault-tolerant systems can also be partially available (also known as performance degradation). This fact is taken into account by *performability* [52], which combines reliability, availability and performance quality attributes. The quantification of performability rewards the system for the performance that is delivered not only during its normal operational time but also during partial failures and their recovery [52]. Essentially, it measures how well the system performs in the presence of failures over a specified period of time.

2.1.2 Dependability Means

To prevent a failure, the chain of dependability threats as shown in Figure 2.1 must be broken. This is possible through *i*) preventing occurrence of faults, *ii*) removing existing faults, or *iii*) tolerating faults. In the last approach, we accept that faults may occur but we deal with their consequences before they lead to a failure, if possible. *Error detection* is the first necessary step for fault tolerance. In addition, detected errors must be *recovered*. Based on [3], Figure 2.2 depicts the dependability means and the features of fault tolerance as a simple *feature diagram*.

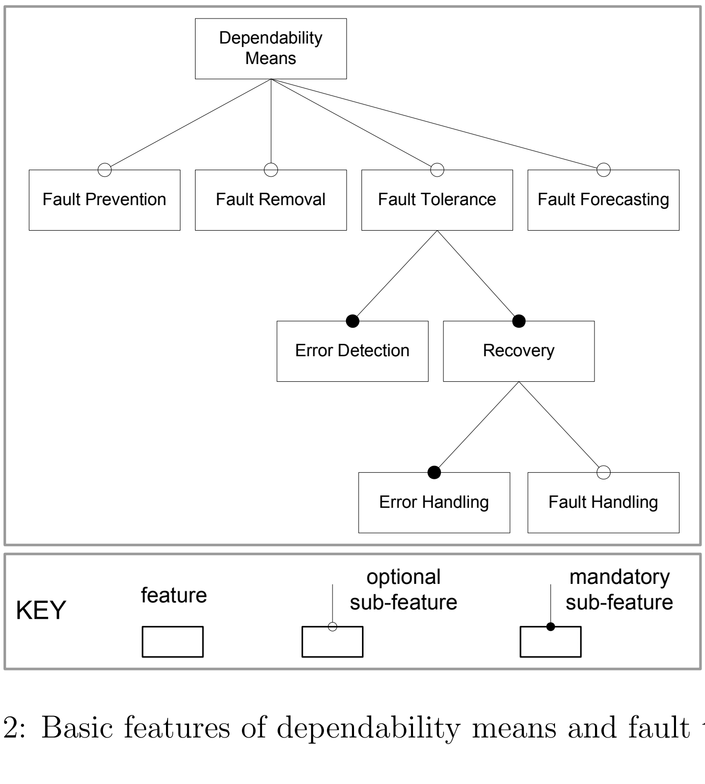


Figure 2.2: Basic features of dependability means and fault tolerance

A feature diagram is a tree, where the root element represents the domain or concept.

The other nodes represent its features. The solid circles indicate mandatory features (i.e. the feature is required if its parent feature is selected). The empty circles indicate optional features. Mandatory features that are connected through an arc decorated edge (See Figure 2.3) are alternatives to each other (i.e. exactly one of them is required if their parent feature is selected).

In addition to *fault prevention*, *fault removal* and *fault tolerance*, *fault forecasting* is also included in Figure 2.2 as a dependability means. Fault forecasting aims at evaluating the system behavior with respect to fault occurrence or activation [3]. Figure 2.2 also shows the two mandatory features of fault tolerance: error detection and recovery. Recovery has one mandatory feature, *error handling*, which eliminates errors from the system state [3]. The *fault handling* feature prevents faults from being activated again. This requires further features such as *diagnosis*, which reveals and localizes the cause(s) of error(s) [3]. Diagnosis can also enable a more effective error handling. If the cause of the error is localized, the recovery procedure can take actions concerning the associated components without impacting the other parts of the system and related system functionality.

2.1.3 Fault Tolerance and Error Handling

When faults manifest themselves during system operations, fault tolerance techniques provide the necessary mechanisms to detect and recover from errors, if possible, before they propagate and cause a system failure. Error recovery is generally defined as the action, with which the system is set to a correct state from an erroneous state [3]. The domain of recovery is quite broad due to different type of faults (e.g. transient, permanent) to be tolerated and different requirements (e.g. cost-effectiveness, high performance, high availability) imposed by different type of systems (e.g. safety-critical systems, consumer electronics). Figure 2.3 shows a partial view of error handling features for fault tolerance. We have derived the features of the recovery domain through a domain analysis based on the corresponding literature [3, 29, 36, 58]

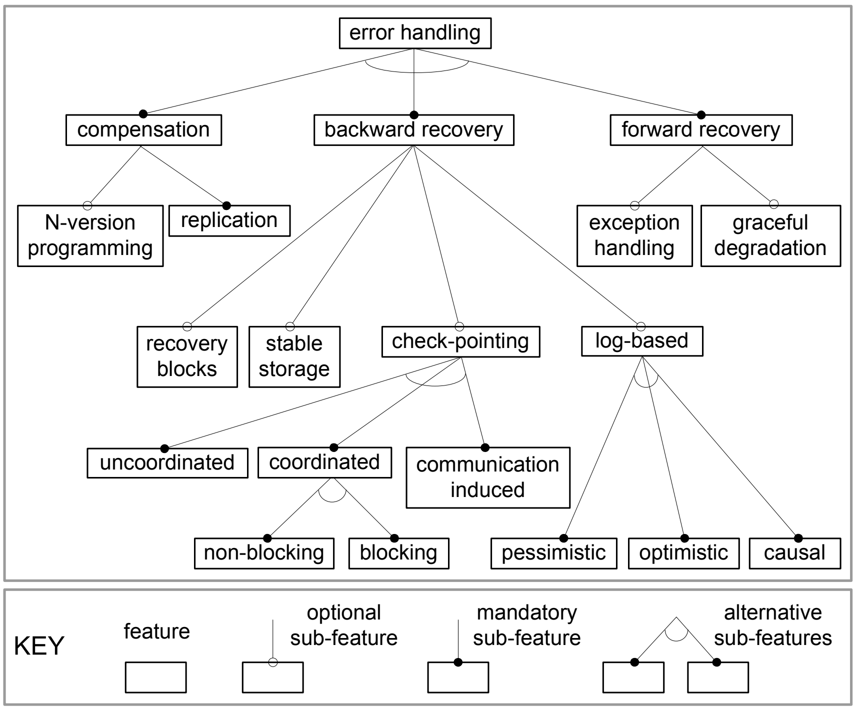


Figure 2.3: A partial view of error handling features for fault tolerance

As shown in Figure 2.3, error handling can be organized into three categories; *compensation*, *backward recovery* and *forward recovery* [3]. Compensation means that the system continues to operate without any loss of function or data in case of an error. This requires replication of system functionality and data. *N-version programming* is a compensation technique, where N independently developed functionally equivalent versions of a software are executed in parallel. All the outputs of these versions are compared to determine the correct, or best output, if one exists [81]. Backward recovery (i.e. rollback) puts the system in a previous state, which was known to be error free. *Recovery blocks* approach uses multiple versions of a software for backward recovery. After the execution of the first version, the output is tested. If the output is not acceptable, the state of the system is rolled back to the state before the first version is executed. Similarly, several versions are executed and tested sequentially until the output is acceptable [81]. The system fails if no acceptable output is obtained after all the versions are tried. Restarting a system is also an example for backward recovery, in which the system is put back to its initial state. Backward recovery can employ different features for saving data to be restored after recovery (i.e. *check-pointing*) or for saving messages and events to replay them after recovery (i.e. *log-based recovery*). In either case, a *stable storage* is required to store recovery-related information (data, messages etc.). Forward recovery (i.e. rollforward) puts the system in a new state to recover from an error.

Exception handling is an example forward recovery technique, where the execution is transfered to the corresponding handler when an exception occurs. *Graceful degradation* [107] is a forward recovery approach that puts the system in a state with reduced functionality and performance.

Different error handling features can be utilized based on the fault assumptions and system characteristics. The granularity of the error handling in recovery can differ as well. In the case of *global recovery*, the recovery mechanism can take actions on the system as a whole (e.g. restart the whole system). In the case of *local recovery*, erroneous parts can be isolated and recovered while the rest of the system is available. Thus, a system with local recovery can provide a higher system availability to its users in case of component failures.

2.2 Software Architecture Design and Analysis

A *software architecture* for a program or computing system consists of the structure or structures of that system, which comprise elements, the externally visible properties of those elements, and the relationships among them [5].

Software architecture represents a common abstraction of a system [5] and as such it forms a basis for mutual understanding and communication among architects, developers, system engineers and anybody who has an interest in the construction of the software system. As one of the earliest artifact of the software development life cycle, software architecture embodies early design decisions, which impacts the system's detailed design, implementation, deployment and maintenance. Hence, it must be carefully documented and analyzed. Software architecture also promotes large-scale reuse by transferring architectural models across systems that exhibit common quality attributes and functional requirements [5].

In the following subsections, we introduce basic techniques and concepts that are used for *i*) describing architectures *ii*) analyzing quality properties of an architecture and *iii*) achieving or supporting qualities in the architecture design.

2.2.1 Software Architecture Descriptions

An *architecture description* is a collection of documents to describe a system's architecture [78]. The IEEE 1471 standard [78] is a recommended practice for architectural description of software-intensive systems. It introduces a set of concepts and relations among them as depicted in Figure 2.4 with a UML (The Unified Modeling

Language) [100] diagram. Hereby, the key concepts are marked with bold rectangles and two types of relations are defined: *association* and *aggregation*. Associations are labeled with a *role* and *cardinality*. For example, Figure 2.4 shows that a concern *is important to* (the role) *1 or more* (the cardinality) stakeholders and a stakeholder *has 1 or more* concerns. Aggregations are identified with a diamond shape at the end of an edge and they represent part-whole relationships. For example, Figure 2.4 shows that a view *is a part of* an architectural description.

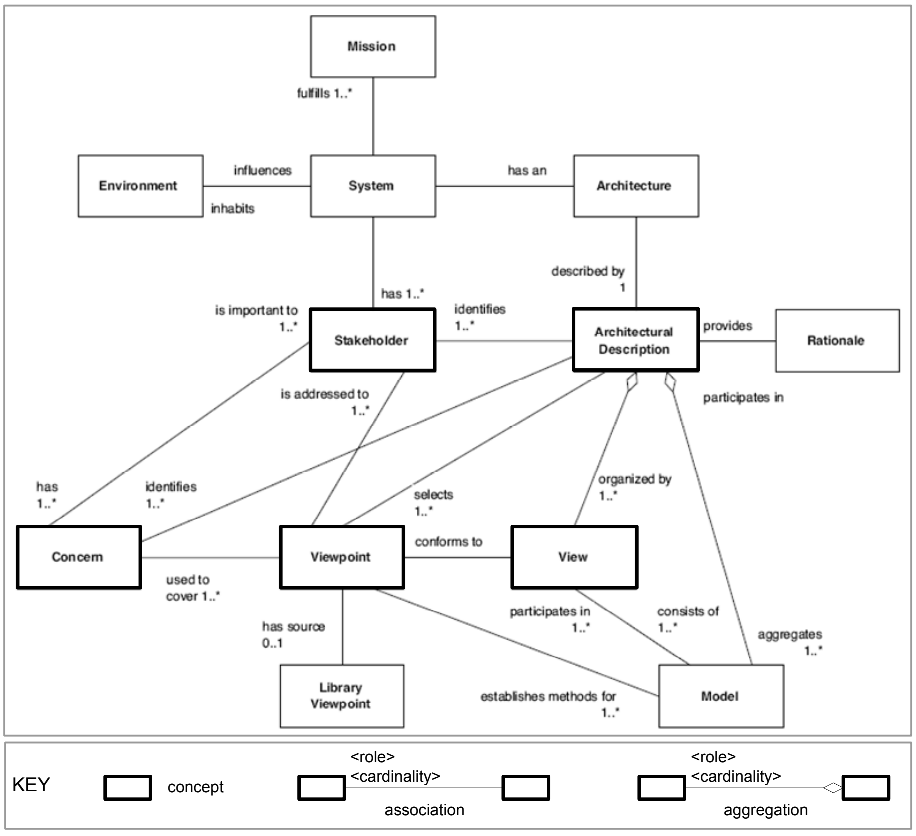


Figure 2.4: Basic concepts of architecture description (IEEE 1471 [78])

The people or organizations that are interested in the construction of the software system are called *stakeholders*. These might include, for instance, end users, architects, developers, system engineers and maintainers. A *concern* is an interest, which pertain to the systems development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders. Stakeholders may have different, possibly conflicting, concerns that they wish the system to provide or optimize. These might include, for instance, certain run-time behavior, performance, reliability and evolvability. A *view* is a representation of the whole system from the perspective of a related set of concerns. A *viewpoint* is a specification of the conventions for constructing and using a view.

To summarize the key concepts as described in the IEEE 1471 framework:

- A system has an architecture.
- An architecture is described by one or more architectural descriptions.
- An architectural description selects one or more viewpoints.
- A viewpoint covers one or more concerns of stakeholders.
- A view conforms to a viewpoint and it consists of a set of models that represent one aspect of an entire system.

This conceptual framework provides a set of definitions for key terms and outlines the content requirements for describing a software architecture. However, it does not standardize or put restrictions to how an architecture is designed and how its description is produced. There are several software design processes and architecture design methods proposed in the literature, such as, the Rational Unified Process [64], Attribute-Driven Design [5] and Synthesis-Based Software Architecture Design [115]. Software architecture design processes and methods are out of cope of this thesis.

Another issue that is not standardized by IEEE 1471 [78] is the notation or format that is used for describing architectures. UML [103] is an example standard notation for modeling object-oriented designs, which can be utilized for describing architectures as well. Similarly, several *architecture description languages* (ADLs) have been introduced as modeling notations to support architecture-based development. There have been both general-purpose and domain-specific ADLs proposed. Some ADLs are designed to have a simple, understandable, and possibly graphical syntax, but not necessarily formally defined semantics. Some other ADLs encompass formal syntax and semantics, supported by powerful analysis tools, model checkers, parsers, compilers and code synthesis tools [82].

2.2.2 Software Architecture Analysis

Software architecture forms one of the key artifacts in software development life cycle since it embodies early design decisions. Accordingly, it is important that the architecture design supports the required qualities of a software system. Software architecture analysis helps to predict the risks and the quality of a system before it is built, thereby reducing unnecessary maintenance costs. On the other hand, usually it is also necessary to evaluate the architecture of a legacy system if it is subject to major modification, porting or integration with other systems [5]. Software architecture constitutes an abstraction of the system, which enables to suppress the unnecessary details and focus only on the relevant aspects for analysis. Basically, there are two complementary software architecture analysis techniques: *i)* *questioning techniques* and *ii)* *measuring techniques* [5].

Questioning techniques use scenarios, questionnaires and check-lists to review how the architecture responds to various situations [19]. Most of the software architecture analysis methods that are based on questioning techniques use scenarios for evaluating architectures [31]. These methods take as input the architecture design and estimate the impact of predefined scenarios on it to identify the potential risks and the sensitivity points of the architecture. Questioning techniques sometimes employ measurements as well but these are mostly intuitive estimations relying on hypothetical models without formal and detailed semantics.

Measuring techniques use architectural metrics, simulations and static analysis of formal architectural models [82] to provide quantitative measures of qualities such as performance and availability. The type of analysis depends on the underlying semantic model of the ADL, where usually a quality model is applied, such as queuing networks [22].

In general, measuring techniques provide more objective results compared to questioning techniques. As a drawback, they require the presence of a working artifact (e.g. a prototype implementation, a model with enough semantics) for measurement. On the other hand, questioning techniques can be applied on hypothetical architectures much earlier in the life cycle [5]. However, (possibly quantitative) results of questioning techniques are inherently subjective. In this thesis, we explore both analysis approaches. In Chapter 3, we present SARAH, which is an analysis method based on questioning techniques. In Chapter 5, we present an analysis approach based on measuring techniques.

2.2.3 Architectural Tactics, Patterns and Styles

A software architect makes a wide range of design decisions, while designing the software architecture of a system. Depending on the application domain, many of these design decisions are made to provide required functionalities. On the other hand, there are also several design decisions made for supporting a desired quality attribute (e.g. to use redundancy for providing fault tolerance and in turn to increase system dependability). Such architectural decisions are characterized as *architectural tactics* [4]. Architectural tactics are viewed as basic design decisions and building blocks of patterns and styles [5].

An *architectural pattern* is a description of element and relation types together with a set of constraints on how they may be used [5]. The term *architectural style* is also used for describing the same concept. Similar to Object-Oriented design patterns [41], architectural patterns/styles provide a common design vocabulary (e.g. clients and servers, pipes and filters, etc.) [12]. They capture recurring idioms, which constrain the design of the system to support certain qualities [109]. Many styles are also equipped with semantic models, analysis tools and methods that enable style-specific analysis and property checks.

Chapter 3

Scenario-Based Software Architecture Reliability Analysis

To select and apply appropriate fault tolerance techniques, we need to analyze potential system failures and identify architectural elements that cause system failures. In this chapter, we propose the Software Architecture Reliability Analysis Approach (SARAH), which prioritizes failure scenarios based on user perception and provides an early software reliability analysis of the architecture design. It is a scenario-based software architecture analysis method that benefits from mature reliability engineering techniques, FMEA and FTA.

The chapter is organized as follows. In the following two sections, we introduce background information on scenario-based software architecture analysis, FMEA and FTA. In section 3.3, we present SARAH and illustrate it for analyzing reliability of the software architecture of the next release of a Digital TV. We conclude the chapter after discussing lessons learned and related work in sections 3.4 and 3.5, respectively.

3.1 Scenario-Based Software Architecture Analysis

Scenario-based software architecture analysis methods take as input a model of the architecture design and measure the impact of predefined scenarios on it to identify the potential risks and the sensitivity points of the architecture [31]. Different analysis methods use different type of scenarios (e.g. usage scenarios [19], change scenarios [7]) depending on the quality attributes that they focus on. Some methods define scenarios just as brief descriptions, while some other methods define them in a more structured way with annotations [19].

Software Architecture Analysis Method (SAAM) can be considered as the first scenario-based architecture analysis method. It is simple, practical and a mature method, which has been validated in various cases studies [19]. Most of the other scenario-based analysis methods are proposed as extensions to SAAM or in some way they adopt the concepts used in this method [31]. The basic activities of SAAM are illustrated with a UML [100] activity diagram in Figure 3.1. The filled circle is the starting point and the filled circle with a border is the ending point. The rounded rectangles represent activities and arrows (i.e. flows) represent transitions between activities. The beginning of parallel activities are denoted with a black bar with one flow going into it and several leaving it. In the following, SAAM activities as depicted in Figure 3.1 are explained.

- *Describe architectures*: The candidate architecture designs are described, which include the systems' computation/data components and their relationships.
- *Define scenarios*: Scenarios are developed for stakeholders to illustrate the kinds of activities the system must support (usage scenarios) and the anticipated changes that will be made to the system over time (change scenarios).
- *Classify/Prioritize scenarios*: Scenarios are prioritized according to their importance as defined by the stakeholders.
- *Individually evaluate indirect scenarios*: Scenarios that can be directly supported by the architecture are called *direct scenarios*. Scenarios that require the redesign of the architecture are called *indirect scenarios*. The required changes for the architecture in case of indirect scenarios are attributed to the fact that the architecture has not been appropriately designed to meet the given requirements. For each indirect scenario the required changes to the architecture are listed and the cost for performing these changes is estimated.

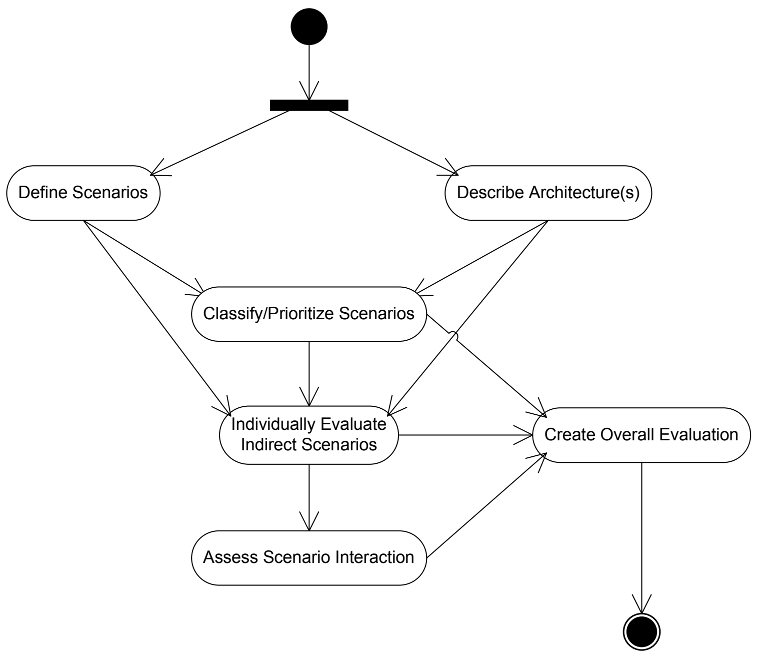

Define Scenarios
Describe Architecture(s)
Classify/Prioritize Scenarios
Individually Evaluate Indirect Scenarios
Create Overall Evaluation
Assess Scenario Interaction

Figure 3.1: SAAM activities [19]

- *Assess scenario interaction*: Determining scenario interaction is a process of identifying scenarios that affect a common set of components. Scenario interaction measures the extent to which the architecture supports an appropriate separation of concerns. Semantically close scenarios should interact at the same component. Semantically distinct scenarios that interact point out a wrong decomposition.

- *Create overall evaluation*: Each scenario and the scenario interactions are weighted in terms of their relative importance and this weighting determines an overall ranking.

SAAM was originally developed to analyze the modifiability of an architecture [19]. Later, numerous scenario-based architecture analysis methods have been developed each focusing on a particular quality attribute or attributes [31]. For example, SAAMCS [75] has focused on analyzing complexity of an architecture, SAAMER [77] on reusability and evolution, ALPSM [8] on maintainability, ALMA [7] on modifiability, ESAAMI [84] on reuse from existing component libraries, and ASAAM [116] on identifying aspects for increasing maintainability.

Hereby, it is implicitly assumed that scenarios correspond to the particular quality attributes that need to be analyzed. Some methods such as SAEM [33] and ATAM [19] have considered the need for a specific quality model for deriving the corresponding scenarios. ATAM has also addressed the interactions among multiple quality attributes and trade-off issues emerging from these interactions.

In this chapter, we propose a *reliability* analysis method, which uses *failure scenarios* for analysis. We define a failure scenario model that is based on the established Failure Modes and Effects Analysis method (FMEA) in the reliability engineering domain as explained in the next subsection.

3.2 FMEA and FTA

Failure Modes and Effects Analysis method (FMEA) [99] is a well-known and mature reliability analysis method for eliciting and evaluating potential risks of a system systematically. The basic operations of the method are *i*) to question the ways that each component fails (failure modes) and *ii*) to consider the reaction of the system to these failures (effects). The analysis results are organized by means of a worksheet, which comprises information about each failure mode, its causes, its local and global effects (concerning other parts of the product and the environment) and the associated component. Failure Modes, Effects and Criticality Analysis (FMECA) extends FMEA with severity and probability assessments of failure occurrence. A simplified FMECA worksheet template is presented in Figure 3.2.

System:	*Car Engine*
Date:	*10-10-2000*
Compiled by:	*J. Smith*
Approved by:	*D. Green*

ID	Item ID	Failure Mode	Failure Causes	Failure Effects	Severity Class
1	*CE5*	*fails to operate*	*Motor shorted*	*Motor overheats and burns*	*V*
2	…	…	…	…	…

Figure 3.2: An example FMECA worksheet based on MIL-STD-1629A [30]

In FMECA, 6 attributes of a failure scenario are identified; *failure id*, *related component*, *failure mode*, *failure cause*, *failure effect* and *severity.* A *failure mode* is defined as the manner in which the element fails. A *failure cause* is the possible cause of a failure mode. A *failure effect* is the (undesirable) consequence of a failure mode. *Severity* is associated with the cost of repair.

FMEA and FMECA can be employed for risk assessment and for discovering potential single-point failures. Systematic analysis increases the insight in the system and the analysis results can be used for guiding the design, its evaluation and improvement. At the downside, the analysis is subjective [97]. Some components failure modes can be overlooked and some information (e.g. failure probability, severity) regarding the failure modes can be incorrectly estimated at early design phases. Since these techniques focus on individual components at a time, combined effects and coordination failures can also be missed. In addition, the analysis is effort and time consuming.

FMEA is usually applied together with *Fault Tree Analysis (FTA)* [34]. FTA is based on a graphical model, *fault tree*, which defines causal relationships between faults. An example fault tree can be seen in Figure 3.3.

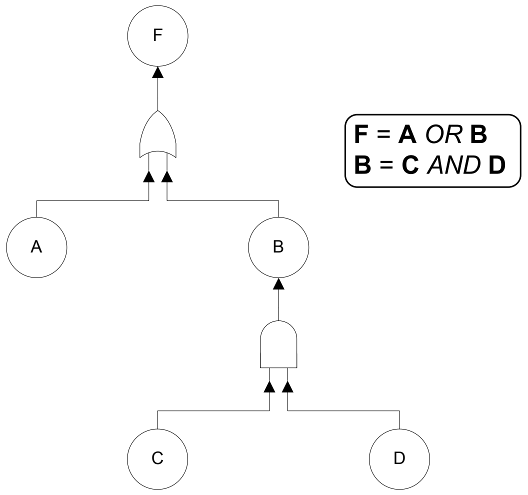


Figure 3.3: An example fault tree

The top node (i.e. root) of the fault tree represents the system failure and the leaf nodes represent faults. Faults, which are assumed to be provided, are defined as undesirable system states or events that can lead to a system failure. The nodes of the fault tree are interconnected with logical connectors (e.g. AND, OR gates) that infer propagation and contribution of faults to the failure. Once the fault tree is constructed, it can be processed in a bottom-up manner to calculate the probability that a failure would take place. This calculation is done based on the probabilities of fault occurrences and interconnections between the faults and the failure [34]. Additionally, the tree can be processed in a top-down manner for diagnosis to determine the potential faults that may cause the failure.

3.3 SARAH

We propose the *Software Architecture Reliability Analysis (SARAH)* approach that benefits from both reliability analysis and scenario-based software architecture analysis to provide an early reliability analysis of next product releases. SARAH defines the notion of failure scenario model that is inspired from FMEA. *Failure scenarios* define potential component failures in the software system and they are used for deriving a fault tree set (FTS). Similar to a fault tree in FTA, FTS shows the causal and logical connections among the failure scenarios.

To a large extent SARAH integrates the best practices of the conventional and stable reliability analysis techniques with the scenario-based software architecture analysis approaches. Besides this, SARAH provides another distinguishing property by focusing on user perceived reliability. Conventional reliability analysis techniques prioritize failures according to how serious their consequences are with respect to safety. In SARAH, the prioritization and analysis of failure scenarios are based on user perception [28]. The structure of FTS and related analysis techniques are also adapted accordingly.

SARAH results in a failure analysis report that defines the sensitive elements of the architecture and provides information on the type of failures that might frequently happen. The reliability analysis forms the key input to identify architectural tactics ([4]) for adjusting the architecture and improving its dependability, which forms the last phase in SARAH. The approach is illustrated using an industrial case for analyzing user-perceived reliability of future releases of Digital TVs. In the following subsection, we present this industrial case, in which a Digital TV architecture is introduced. This example will be used throughout the remainder of the section, where the activities of SARAH are explained and illustrated. As such while explaining the approach we also discuss our experience and obstacles in applying the approach.

3.3.1 Case Study: Digital TV

A conceptual architecture of Digital TV (DTV) is depicted in Figure 3.4, which will be referred throughout the section. The design mainly comprises two layers. The bottom layer, namely the streaming layer, involves modules taking part in streaming of audio/video information. The upper layer consists of applications, utilities and modules that control the streaming process. In the following, we briefly explain some of the important modules that are part of the architecture. For brevity, the modules for decoding and processing audio/video signals are not explained here.

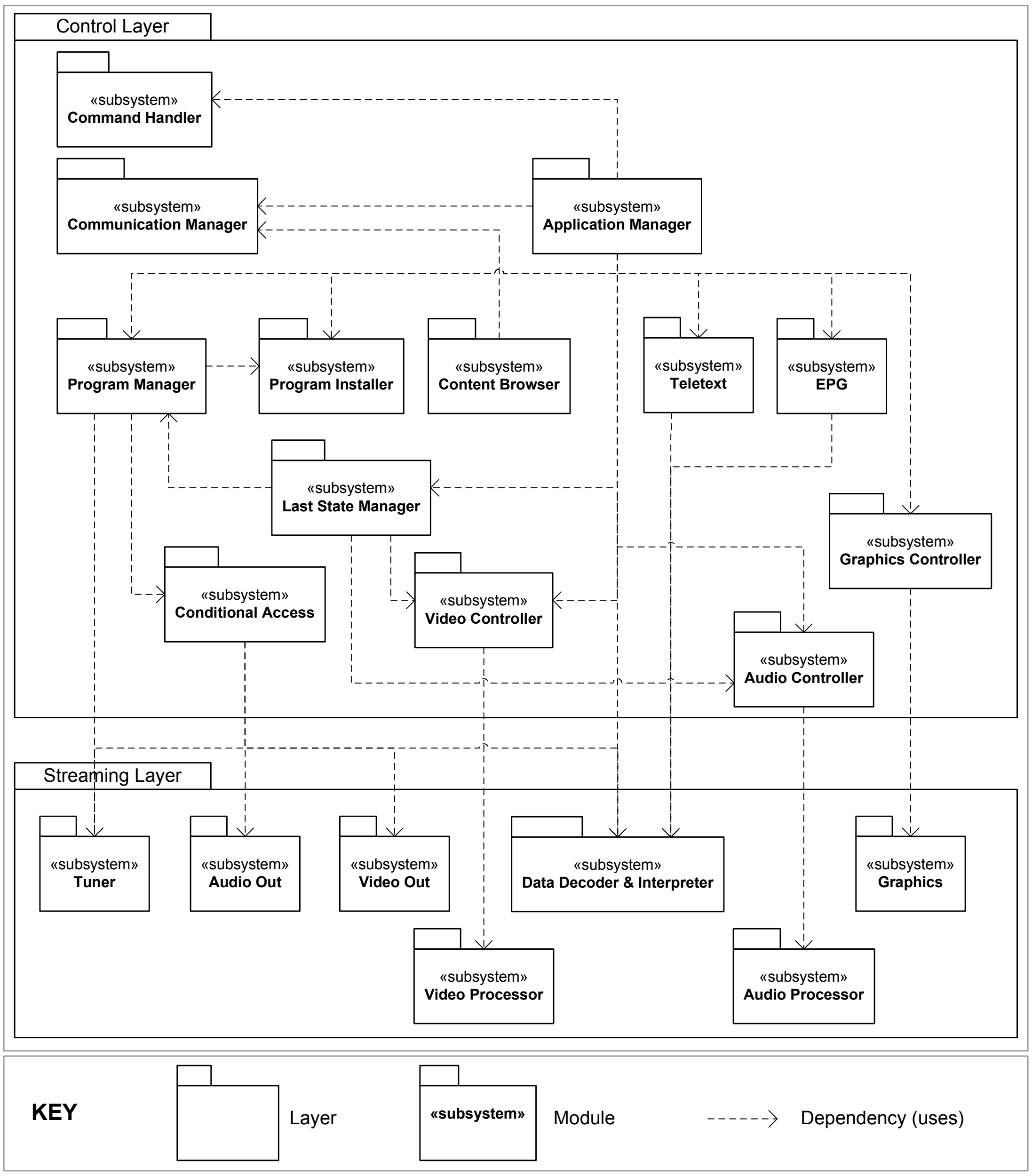

Control Layer
«subsystem» Command Handler
«subsystem» Communication Manager
«subsystem» Application Manager
«subsystem» Program Manager
«subsystem» Program Installer
«subsystem» Content Browser
«subsystem» Teletext
«subsystem» EPG
«subsystem» Last State Manager
«subsystem» Graphics Controller
«subsystem» Conditional Access
«subsystem» Video Controller
«subsystem» Audio Controller
Streaming Layer
«subsystem» Tuner
«subsystem» Audio Out
«subsystem» Video Out
«subsystem» Data Decoder & Interpreter
«subsystem» Graphics
«subsystem» Video Processor
«subsystem» Audio Processor
KEY
Layer
«subsystem»
Module
Dependency (uses)

Figure 3.4: Conceptual Architecture of DTV

- *Application Manager (AMR)*, located at the top middle of the figure, initiates and controls execution of both resident and downloaded applications in the system. It keeps track of application states, user modes and redirects commands/information to specific applications or controllers accordingly.
- *Audio Controller (AC)*, located at the bottom right of the figure, controls

audio features like volume level, bass and treble based on commands received from AMR.

- *Command Handler (CH)*, located at the top left of the figure, interprets externally received signals (i.e. through keypad or remote control) and sends corresponding commands to AMR.
- *Communication Manager (CMR)*, located at the top left of the figure, employs protocols for providing communication with external devices.
- *Conditional Access (CA)*, located at the bottom left of the figure, authorizes information that is presented to the user.
- *Content Browser (CB)*, located at the middle of the figure, presents and provides navigation of content residing in a connected external device.
- *Electronic Program Guide (EPG)*, located at the middle right of the figure, presents and provides navigation of electronic program guide regarding a channel.
- *Graphics Controller (GC)*, located at the bottom right of the figure, is responsible for generation of graphical images corresponding to user interface elements.
- *Last State Manager (LSM)*, located at the middle of the figure, keeps track of last state of user preferences such as volume level and selected program.
- *Program Installer (PI)*, located at the middle of the figure, searches and registers programs together with channel information (i.e. frequency).
- *Program Manager (PM)*, located at the middle left of the figure, tunes to a specific program based on commands received from AMR.
- *Teletext (TXT)*, located at the middle of the figure, handles acquisition, interpretation and presentation of teletext pages.
- *Video Controller (VC)*, located at the bottom middle of the figure, controls video features like scaling of the video frames based on commands received from AMR.

3.3.2 The Top-Level Process

For understanding and predicting quality requirements of the architectural design [4], Bachman et al. identify four important requirements: *i*) provide a specification of the quality attribute requirements, *ii*) enumerate the architectural decisions to achieve the quality requirements, *iii*) couple the architectural decisions to the quality attribute requirements, and *iv*) provide the means to compose the architectural decisions into a design. SARAH is in alignment with these key assumptions. The focus in SARAH is the specification of the reliability quality attribute, the analysis of the architecture based on this specification and the identification of architectural tactics to adjust the architecture.

The steps of SARAH are presented as a UML activity diagram in Figure 3.5. The approach consists of three basic processes: *i*) *Definition ii*) *Analysis* and *iii*) *Adjustment*. In the definition process the architecture, the failure domain model, the failure scenarios, the fault trees and the severity values for failures are defined. Based on this input, in the analysis process, an architectural level analysis and an architectural element level analysis are performed. The results are presented in the failure analysis report. The failure analysis report is used in the adjustment process to identify the architectural tactics and adapt the software architecture. In the following subsections the main steps of the method will be explained in detail using the industrial case study.

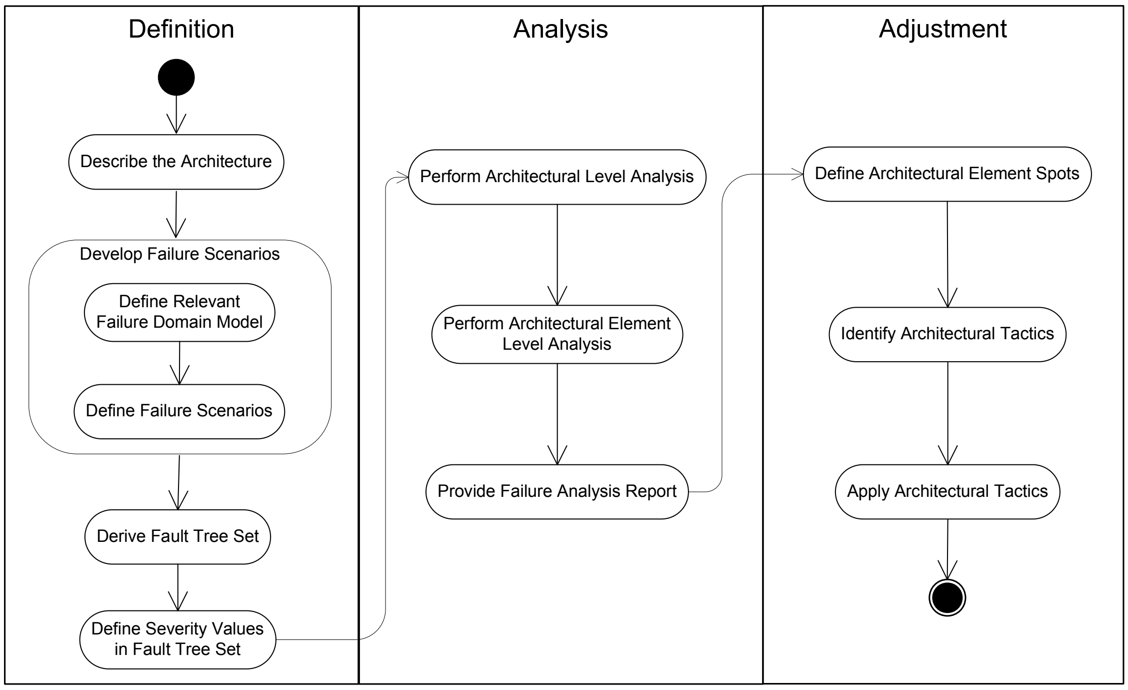

Definition
Analysis
Adjustment
Describe the Architecture
Develop Failure Scenarios
Define Relevant Failure Domain Model
Define Failure Scenarios
Derive Fault Tree Set
Define Severity Values in Fault Tree Set
Perform Architectural Level Analysis
Perform Architectural Element Level Analysis
Provide Failure Analysis Report
Define Architectural Element Spots
Identify Architectural Tactics
Apply Architectural Tactics

Figure 3.5: Activity Diagram of the Software Architecture Analysis Method

3.3.3 Software Architecture and Failure Scenario Definition

Describe the Architecture

Similar to existing software architecture analysis methods SARAH starts with describing the software architecture. The description includes the architectural elements and their relationships. Currently, the method itself does not presume a particular architectural view [18] to be provided but in our project we have basically applied it to the module view. The architecture that we analyzed is depicted in Figure 3.4.

Develop Failure Scenarios

SARAH is a scenario-based architecture analysis method, that is, scenarios are the basic means to analyze the architecture. SARAH defines the concept of *failure scenario* to analyze the architecture with respect to reliability. A failure scenario defines a chain of dependability threats (i.e. fault, error and failure) for a component of the system. To specify the failure scenarios in a uniform and consistent manner a failure scenario template, as defined in Table 3.1 is adopted for specifying failure scenarios.

Table 3.1: Template for Defining Failure Scenarios

FID	A numerical value to identify the failures (i.e. Failure ID)
AEID	An acronym defining the architectural element for which the failure scenario applies (i.e. Architectural Element ID)
Fault	The cause of the failure defining both the description of the cause and its features
Error	Description of the state of the element that leads to the failure together with its features
Failure	The description of the failure, its features, user/element(s) that are affected by the failure

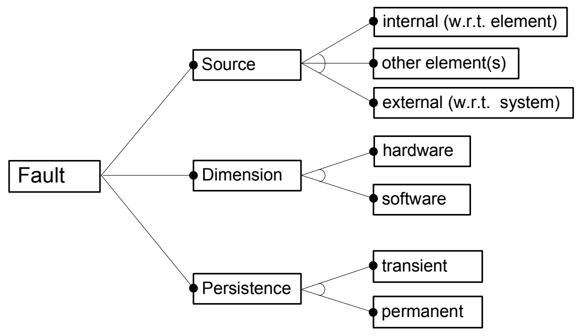
The template is inspired from FMEA [99]. For clarity in SARAH *fault*, *error* and *failure* are used instead of the concepts *failure cause*, *failure mode* and *failure effect*, respectively. In SARAH, failure scenarios are derived in two steps. First the relevant failure domain model is defined, then failure scenarios are derived from this failure domain. The following subsections describe these steps in detail.

Define Relevant Failure Domain Model

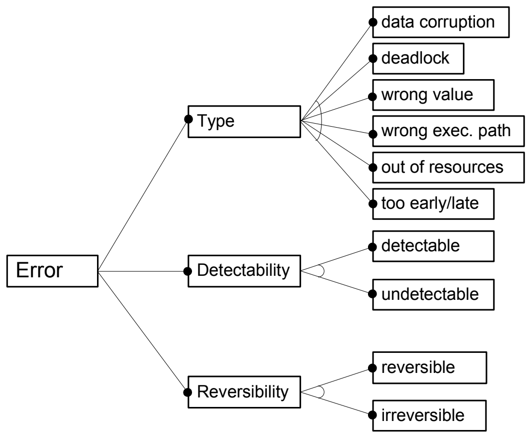
The failure scenario template can be adopted to derive scenarios in an ad hoc manner using free brainstorming sessions. However, it is not trivial to define fault classes, error types or failure modes. Hence, there is a high risk that several potential and relevant failure scenarios are missed or that other irrelevant failure scenarios are included. To define the space of relevant failures SARAH defines relevant domain model for faults, errors and failures using a systematic domain analysis process [2]. These domain models provide a first scoping of the potential scenarios. In fact, several researchers have already focused on modeling and classifying failures. Avizienis et al., for example, provide a nice overview of this related work and provide a comprehensive classification of faults, errors and failures [3]. The provided domain classification by Avizienis et al., however, is rather broad, and one can assume that for a given reliability analysis project not all the potential failures in this overall domain are relevant. Therefore, the given domain is further scoped by focusing only on the faults, errors and failures that are considered relevant for the actual project. Figure 3.6, for example, defines the derived domain model that is considered relevant for our project.

In Figure 3.6(a), a feature diagram is presented, where faults are identified according to their source, dimension and persistence. In SARAH, failure scenarios are defined per architectural element. For that reason, the source of the fault can be either *i*) internal to the element in consideration, *ii*) caused by other element(s) of the system that interact(s) with the element in consideration or *iii*) caused by external entities with respect to the system. Faults could be caused by software or hardware, and be transient or persistent. In Figure 3.6(b), the relevant features of an error are shown, which comprise the type of error together with its detectability and reversibility properties. Figure 3.6(c) defines the features for failures, which includes the features type and target. The target of a failure defines what is/are affected by the failure. In this case, the target can be the user or other element(s) of the system.

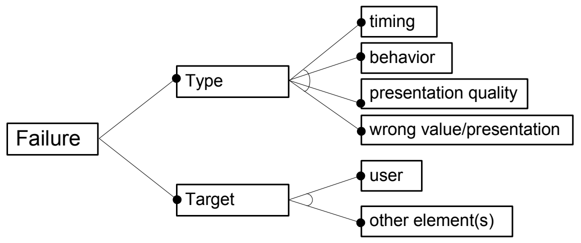
The failure domain model of Figure 3.6 has been derived after a thorough domain analysis and in cooperation with the domain experts in the project. In principle, for different project requirements one may come up with a slightly different domain model, but as we will show in the next sections this does not impact the steps in the analysis method itself. The key issue here is that failure scenarios are defined based on the FMEA model, in which their properties are represented by domain models that provide the scope for the project requirements.



(a) Feature Diagram of Fault



(b) Feature Diagram of Error



(c) Feature Diagram of Failure

Figure 3.6: Failure Domain Model

Define Failure Scenarios

The domain model defines a system-independent specification of the space of failures that could occur. The number and type of failure scenarios are implicitly defined by the failure domain model, which defines the scope of the relevant scenarios. In the fault domain model (feature diagram in Figure 3.6(a)), for example, we can define faults based on three features, namely *Source*, *Dimension* and *Persistence*. The feature *Source* can have 3 different values, the features *Dimension* and *Persistence* 2 values. This means that the fault model captures $3 \times 2 \times 2 = 12$ different faults. Similarly, from the error domain model we can derive $6 \times 2 \times 2 = 24$ different errors, and $4 \times 2 = 8$ different failures are captured by the failure domain model. Since a scenario is a composition of selection of features from the failure domain model, we can state that for the given failure domain model in Figure 3.6 in theory, $12 \times 8 \times 24 = 2304$ failure scenarios can be defined. However, not all the combinations instantiated from the failure domain are possible in practice. For example, in case the error type is "too late" then the error cannot be "reversible". If the failure type is "presentation quality" then the failure target can only be "user". To specify such kind of constraints for the given model in Figure 3.6 usually *mutex* and *requires* predicates are used [66]. The given two example constraints are, for example, specified as follows:

```
Error.type.too late mutex
Error.type.reversibility.reversible

Failure.type.presentation quality requires
Failure.target.user
```

Once the failure domain model together with the necessary constraints have been specified we can start defining failure scenarios for the architectural elements. For each architectural element we check the possible failure scenarios and define an additional description of the specific fault, error or failure. Figure 3.7 provides, for example, a list of nine selected failure scenarios that have been derived for the reliability analysis of the DTV. In Figure 3.7 the five elements *FID*, *AEID*, *fault*, *error* and *failure* are represented as columns headings. Failure scenarios are represented in rows.

FID	AEID	Fault	Error	Failure
F1	AMR	*description:* Reception of irrelevant signals. *source*: CH(F4) OR CMR(F6) *dimension*: software *persistence*: permanent	*description:* Working mode is changed when it is not desired. *type*: wrong path *detectability*: undetectable *reversibility*: reversible	*description:* Switching to an undesired mode. *type*: behavior *target*: user
F2	AMR	*description:* Can not acquire information. *source*: CMR(F5) *dimension*: software *persistence*: permanent	*description:* Information can not be acquired from the connected device. *type*: too early/late *detectability*: detectable *reversibility*: irreversible	*description:* Can not provide information. *type*: timing *target*: CB(F3)
F3	CB	*description:* Can not acquire information. *source*: AMR(F2) *dimension*: software *persistence*: permanent	*description:* Information can not be presented due to lack of information. *type*: too early/late *detectability*: detectable *reversibility*: irreversible	*description:* Can not present content of the connected device. *type*: behavior *target*: user
F4	CH	*description:* Software fault. *source*: internal *dimension*: software *persistence*: permanent	*description:* Signals are interpreted in a wrong way. *type*: wrong value *detectability*: undetectable *reversibility*: reversible	*description:* Provide irrelevant information. *type*: wrong value/presentation *target*: AMR(F1)
F5	CMR	*description:* Protocol mismatch. *source*: external *dimension*: software *persistence*: permanent	*description:* Communication can not be sustained with the connected device. *type*: too early/late *detectability*: detectable *reversibility*: irreversible	*description:* Can not provide information. *type*: timing *target*: AMR(F2)
F6	CMR	*description:* Software fault. *source*: internal *dimension*: software *persistence*: permanent	*description:* Signals are interpreted in a wrong way. *type*: wrong value *detectability*: undetectable *reversibility*: reversible	*description:* Provide irrelevant information. *type*: wrong value/presentation *target*: AMR(F1)
F7	DDI	*description:* Reception of out-of-spec signals. *source*: external *dimension*: software *persistence*: permanent	*description:* Scaling information can not be interpreted from meta-data. *type*: wrong value *detectability*: detectable *reversibility*: reversible	*description:* Can not provide data. *type*: wrong value/presentation *target*: VP(F8)
F8	VP	*description:* Inaccurate scaling ratio information. *source*: DDI(F7) AND VC(F9) *dimension*: software *persistence*: permanent	*description:* Video image can not be scaled appropriately. *type*: wrong value *detectability*: undetectable *reversibility*: reversible	*description:* Provide distorted video image. *type*: presentation quality *target*: user
F9	VC	*description:* Software fault. *source*: internal *dimension*: software *persistence*: permanent	*description:* Correct scaling ratio can not be calculated from the video signal. *type*: wrong value *detectability*: detectable *reversibility*: reversible	*description:* Provide inaccurate information. *type*: wrong value/presentation *target*: VP(F8)

Figure 3.7: Selected failure scenarios for the analysis of the DTV architecture

The failure ids represented by FID do not have a specific ordering but are only used to identify the failure scenarios. The column AEID includes acronyms of names of architectural elements to which the identifiers refer. The type of the architectural elements that are analyzed can vary depending on the architectural view utilized [18]. In case of a deployment view, for instance, the architectural elements that will be analyzed will be the nodes. For component and connector view, the architectural elements will be components and connectors. In this paper, we focused on module view of the architecture, where the architectural elements are the implementation units (i.e. modules). In principle, when an architectural element is represented as a first-class entity in the model and it affects the failure behavior, then it can be used in SARAH for analysis.

The columns *Fault*, *Error* and *Failure* describe the specific faults, errors and failures. Note that the separate features of the corresponding domain models are represented as keywords in the cells. For example, *Fault* includes the features *source*, *dimension* and *persistence* as defined in Figure 3.6(a), and likewise these are represented as keywords. Besides the different features, each column also includes the keyword *description*, which is used to explain the domain specific details of the failure scenarios. Typically these descriptions are obtained from domain experts. The columns *Fault* and *Failure* include the fields *source* and *target* respectively. Both of these refer to failure scenarios of other architectural elements. For example, in failure scenario F2, the fault source is defined as CMR(F5), indicating that the fault in F2 occurs due to a failure in CMR as defined in F5. The source of the fault can be caused by a combination of failures. This is expressed by logical connectives. For example, the source of F1 is defined as CH(F4) OR CMR(F6) indicating that F1 occurs due to a failure in CH as defined in F4 or due to a failure in CMR as defined in F6.

Derive Fault Tree Set from Failure Scenarios

A close analysis of the failure scenarios shows that they are connected to each other. For example, the failure scenario F1 is caused by the failure scenario F4 or F6. To make all these connections explicit, in SARAH *fault trees* are defined. Typically, a given set of failure scenarios leads to a set of fault trees, which are together defined as a *Fault Tree Set (FTS)*. FTS is a graph $G(V, E)$ consisting of the set of fault trees. G has the following properties.

1. $V = F \cup A$

2. F is the set of failure scenarios each of which is associated with an architectural element.

3. $F_u \subseteq F$ is the set of failure scenarios comprising failures that are perceived by the user (i.e. system failures). Vertices residing in this set constitute root nodes of fault trees.

4. A is the set of gates representing logical connectives.

5. $\forall g \in A$,

   $outdegree(g) = 1 \wedge$

   $indegree(g) \geq 1$

6. $A = A_{AND} \cup A_{OR}$ such that,

   A_{AND} is the set of AND gates

   A_{OR} is the set of OR gates

7. E is the set of directed edges (u, v) where $u, v \in V$.

For the example case, based on the failure scenarios in Figure 3.7 we can derive the FTS as depicted in Figure 3.8.

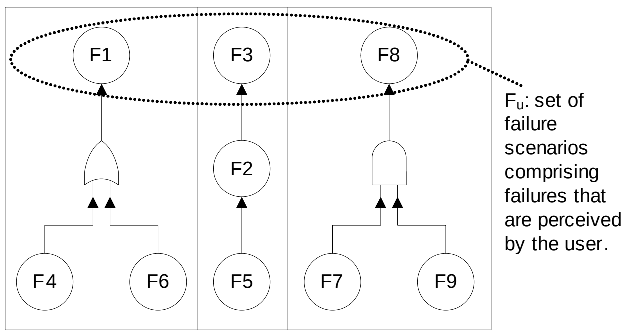

F1
F3
F8
F2
F4
F6
F5
F7
F9
F$_u$: set of failure scenarios comprising failures that are perceived by the user.

Figure 3.8: Fault trees derived from failure scenarios in Figure 3.7

Here, the FTS consists of three fault trees. On the left the fault tree shows that F1 is caused by F4 or F6. The middle column indicates that failure scenario F3 is caused by F2 which is on its turn caused by F5. Finally, in the last column the fault tree shows that F8 is caused by both F7 and F9.

Define Severity Values in Fault Tree Set

As we have indicated before we are focused on failure analysis in the context of consumer electronics domain. As such we need to analyze in particular those failures which have the largest impact on the user perception. For example, a complete black screen will have a larger impact on the user than a temporary distortion in the image. These different user perceptions on the failures have a clear impact on the way how we process the fault trees. Before computing the failure probabilities of the individual leaf nodes, we first assign *severity* values to the root failures based on their impact on the user. In our case the severity values are based on the prioritization of the failure scenarios as defined in Table 3.2.

Table 3.2: Prioritization of severity categories

Severity	**Type**	**Annoyance Description**
1.	very low	User hardly perceives failure.
2.	low	A failure is perceived but not really annoying.
3.	moderate	Annoying performance degradation is perceived.
4.	high	User perceives serious loss of functions.
5.	very high	Basic user-perceived functions fail. System locks up and does not respond.

The severity degrees range from 1 to 5 and are provided by domain experts. For example, the severity values for failures F1, F3 and F8 are depicted in Figure 3.9. Here we can see that F5 has been assigned the severity value 5 indicating that it has a very high impact on the user perception. In fact, the impact of a failure can differ from user to user. In this paper, we consider the average user type. To define user perceived failure severities, an elaborate user model can be developed based on experiments with subjects from different age groups and education levels [28]. We consider the user model as an external input and any such model can be incorporated to the method.

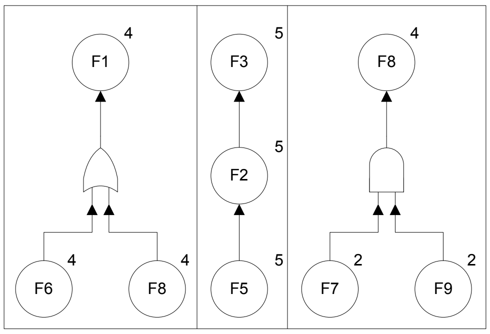


Figure 3.9: Fault trees with severity values

The severity values of the root nodes are used for determining the severity values of the other nodes of the FTS. These values are calculated based on the following equations:

$$\forall u \in F_u, s(u) = s_u \tag{3.1}$$

$$\forall f \in F \wedge f \notin F_u, s(f) = \sum_{\forall v s.t. (f,v) \in E} s(v) \times P(v|f) \tag{3.2}$$

Equation 3.1 defines the assignment of severity values to the root nodes. Equation 3.2 defines the calculation of the severity values for the other nodes. Here, $P(v|f)$ denotes the probability that the occurrence of f will cause the occurrence of v [34]. We multiply this value with the severity of v to calculate the severity of f. According to the probability theory, $P(v|f) = P(v \cap f)/P(f)$. If v is an OR gate, then the output of v will always occur whenever f occurs. That is, $P(v \cap f) = P(f)$. As a result, $P(v|f) = P(f)/P(f) = 1$. This makes sense because $P(v|f)$ denotes the probability that v will occur, provided that f occurs. If we take the OR gate at the left hand side of Figure 3.9 for example, $P(v|F6) = 1$. If we know that F6 occurs, v will definitely occur (the probability is equal to 1). This is also the case for F8. So, $s(F6) = s(F8) = s(F1) = 4$. If v is an AND gate, the situation is different. In this case, $P(v \cap f) = \Pi P(x)$ for all vertices x that is connected to v, including f. Since $P(v|f) = P(v \cap f)/P(f)$, to calculate $P(v|f)$ we need to know $P(x)$ for all x except f. If we take the AND gate at the right hand side of Figure 3.9 for example, $P(v|F7) = P(F7) \times P(F9)/P(F7) = P(F9)$ and similarly $P(v|F7) = P(F9)$. So, $s(F7) = s(F8) \times P(F8|F7) = 4 \times P(F9)$ and $s(F9) = s(F8) \times P(F8|F9) = 4 \times P(F7)$.

To define the final severity values for all the nodes obviously we need to know the probability of each failure. In principle there are three strategies that can be used to determine these required probability values:

- *Using Fixed values*: All probability values can be fixed to a certain value. An example is to assume that each failure will have equal weight and likewise the probabilities of individual failures are basically defined by the AND and OR gates.
- *What-if analysis*: A range of probability values can be considered, where fault probabilities are varied and their effect on the probability of user perceived failures are observed.
- *Measurement of actual failure rate*: The actual failure rate can be measured based on historical data or execution probabilities of elements that are obtained from scenario-based test runs [126].

In SARAH all these three strategies can be used depending on the available knowledge on probabilities. In case the probabilities are not known or they do not have a relevant impact on the final outcome then fixed probability values can be adopted. If probabilities are not equal or have different impacts on the severity values then either the second or third strategy can be used. In the second strategy, the what-if analysis can be useful if no information is available on an existing system. The measurement of actual failure rates is usually the most accurate way to define the severity values. However, the historical data can be missing or not accessible, and deriving execution probabilities is cumbersome.

To identify the appropriate strategy a preliminary *sensitivity analysis* can be performed. In conventional FTA, sensitivity analysis is based on cut-sets in a fault tree and the probability values of fault occurrences [34]. However, this analysis leads to complex formulas and it requires that the probability values are known a priori. In our case we propose the following model (based on [14] and [126]) for estimating the sensitivity with respect to a fault probability even if the probability values are not known.

$$\begin{aligned} root \in F_u, node \in F, \\ \forall n \in F \wedge n \neq node \wedge n \neq root, P(n) = p \qquad (3.3) \\ P(node) = p', P(root) = f(p', p), \end{aligned}$$

$$sensitivity(node) = \int_0^1 (\frac{\partial}{\partial p'} P(root))dp \qquad (3.4)$$

Equations 3.3 and 3.4 show the calculation of the sensitivity of probability of a user perceived failure ($P(root)$) to the probability of a node ($P(node)$) of the fault tree. Here, we assign p' to $P(node)$ and fix the probability values of all the other nodes to p. Thus $P(root)$ turns out to be a function of p and p'. For example, if we are interested in the sensitivity of $P(F8)$ to $P(F9)$, $P(F8) = P(F7) \times P(F9) = p' \times p$. Then, we take a partial derivative of $P(root)$ with respect to p'. This gives the rate of change of $P(root)$ with respect to p' ($P(F9)$). For our example, this will yield to $\partial/\partial p' P(F8) = p$. Finally, the result of the partial derivation is integrated with respect to p for all possible probability values (range from 0 to 1) to calculate the overall impact. For the example case, $\int(\partial/\partial p' P(F8))dp = \int p dp = p^2/2$. So, the result of the integration from 0 to 1 will be 0.5. In this model we use the basic sensitivity analysis approach, where we change a parameter one at a time and fix the others ([126]). Additionally, we use derivation to calculate the rate of change ([14]) and we use integration to take the range of probability values into account. The resulting formulas are simple enough to be computed with spreadsheets.

For the analysis presented here, we skip the sensitivity analysis and assume equal probabilities (i.e. $P(x) = p$ for all x), which simplifies the severity assignment formula for AND gates as $s(v) \times P(v|f) = s(v) \times p^{indegree(v)-1}$. Based on this assumption, the severity calculation for intermediate failures is as shown in the following equation.

$$s(f) = \sum_{\substack{\forall v s.t. \\ (u,v)\in E \wedge \\ (v\in F \vee v \in A_{OR})}} s(v) + \sum_{\substack{\forall v s.t. \\ (u,v)\in E \wedge \\ v\in A_{AND}}} s(v) \times p^{indegree(v)-1} \tag{3.5}$$

In our analysis, we fixed the value of p to be 0.5. In case of F7 and F9, for instance, the severity value is 4/2 because F8 has the severity value of 4 and it is connected to an AND gate with $indegree = 2$. On the other hand, F1 has the assigned severity value of 4 and this is also assigned to the failures F6 or F8 that are connected to F1 through an OR gate. A failure scenario can be connected to multiple gates and other failures, in which the severity value is derived as the sum of the severity values calculated for all these connections.

3.3.4 Software Architecture Reliability Analysis

In our example case, we defined a total of 37 failure scenarios including the scenarios presented in Figure 3.7. We completed the definition process by deriving the corresponding fault tree set and calculating severity values as explained in section 3.3.3

(See Appendix A). The results that were obtained during the definition process are utilized by the analysis process as described in the subsequent sections.

Perform Architecture-Level Analysis

The first step of the analysis process is *architecture-level analysis* in which we pinpoint sensitive elements of the architecture with respect to reliability. Sensitive elements are the elements, which are associated with the majority of the failure scenarios. These include not only the failure scenarios caused by internal faults but also the failure scenarios caused by other elements. In the architectural-level analysis step, we aim at identifying sensitive elements with respect to the most critical failures from the user perception. Later, in the architectural element level analysis, the fault, error types and the actual sources of their failures (internal or other elements) are identified. Sensitive elements provide a starting point for considering the relevant fault tolerance techniques and the elements to improve with respect to fault tolerance (see section 3.3.5). In this way the effort that is provided for reliability analysis is scoped with respect to failures that are directly perceivable by the users.

As a primary and straightforward means of comparison, we consider the percentage of failures (PF) that are associated with elements. For each element c the value for PF is calculated as follows:

$$PF_c = \frac{\#\_of\_failures\_associated\_with\_c}{\#\_of\_failures} \times 100 \tag{3.6}$$

This means that simply all the number of failures related to an element are summed and divided by the total number of failures (in this case 44). The results are shown in Figure 3.10. A first analysis of this figure already shows that the *Application Manager (AMR)* and *Teletext (TXT)* modules have to cope with a higher number of failures than other modules.

This analysis treats all failures equally. To take the severity of failures into account we define the *Weighted Percentage of Failures (WPF)* as given in Equation 3.7.

$$WPF_c = \frac{\sum_{\substack{\forall u \in F s.t. \\ AEID(u)=c}} s(u)}{\sum_{\forall u \in F} s(u)} \times 100 \tag{3.7}$$

For each element, we collect the set of failures associated with them and we add up their severity values. After averaging this value with respect to all failures and

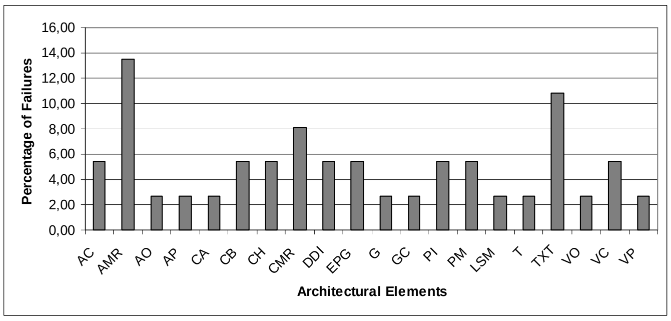


Figure 3.10: Percentage of failure scenarios impacting architectural elements

calculating the percentage, we obtain the WPF value. The result of the analysis is shown in Figure 3.11.

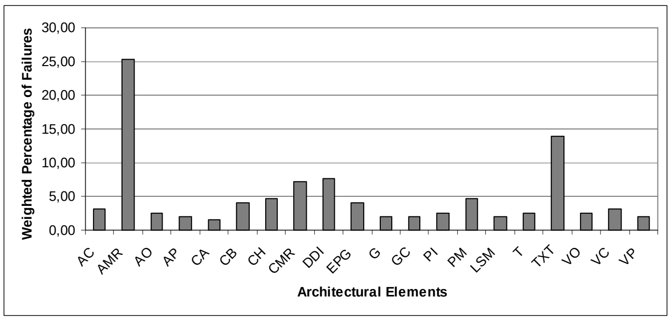


Figure 3.11: Weighted percentage of failure scenarios impacting architectural elements

Although weighted percentage presents different results compared to the previous one, the *Application Manager (AMR)* and *Teletext(TXT)* modules again appears to be very critical.

From the project perspective it is not always possible to focus on the total set of possible failures due to the related cost for fault tolerance. To optimize the cost usually one would like to consider the failures that have the largest impact on the system. For this, in SARAH the architectural elements are ordered in ascending order with respect to their WPF values. The elements are then categorized based on the proximity of their WPF values. Accordingly, elements of the same group have WPF values that are close to each other. The results of this prioritization and grouping are provided in Table 3.3, which also shows the sum of the WPF values of elements for each group. Here we can see that, for example, group 4 consists of two modules AMR and TXT. The reason for this is that their WPF values are the highest and close to each other. The sum of their WPF values is $25 + 14 = 39\%$.

Table 3.3: Architectural elements grouped based on WPF values

Group #	**Modules**	**WPF**
1	AC, AO, AP, CA, G, GC, PI, LSM, T, VO, VC, VP	28%
2	CB, CH, EPG, PM	18%
3	DDI, CMR	15%
4	AMR, TXT	39%

To highlight the difference in impact of the groups of architectural elements we define a Pareto chart as presented in Figure 3.12.

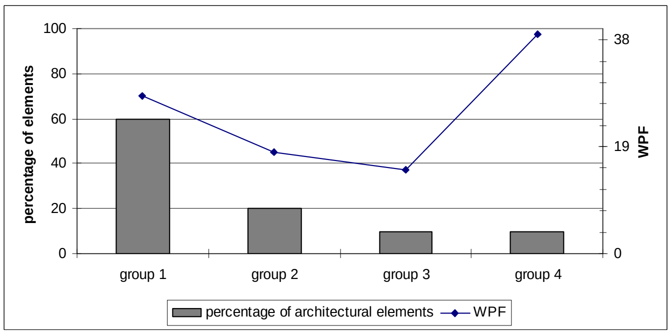


Figure 3.12: Pareto Chart showing the largest impact of the smallest set of architectural elements

In the Pareto chart, the groups of architectural elements shown in Table 3.3 are ordered along the x-axis with respect to the number of elements they include. The percentage of elements that each group includes is depicted with bars. The y-axis on the left hand shows the percentage values from 0 to 100 and is used for scaling the percentages of architectural elements whereas the y-axis on the right hand side scales WPF values. The plotted line represents the WPF value for each group. In the figure we can, for example, see that group 4 (consisting of two elements) represents 10% of all the elements but has a WPF of 39%. The chart helps us in this way to focus on the most important set of elements which are associated with the majority of the user perceived failures.

Perform Architectural Element Level Analysis

The architectural level analysis provides only an analysis of the impact of failure scenarios on the given system. However, for error handling and recovery it is also necessary to define the type of failures that might affect the identified sensitive elements. This is analyzed in the architectural element level analysis in which the features of faults, errors and failures that impact an element are determined. For the example case, in the architectural-level analysis it appeared that elements residing in the 4^{th} group (see Table 3.3) had to deal with largest set of failure scenarios. Therefore, in *architectural element level analysis*, we will focus on members of this group, namely *Application Manager* and *Teletext* modules.

Following the derivation of the set of failure scenarios impacting an element, we group them in accordance with the features presented in Figure 3.6. This grouping results in the distribution of fault, error and failure categories of failure scenarios associated with the element.

For example, the results obtained for *Application Manager* and *Teletext* modules are shown in Figure 3.13 and Figure 3.14, respectively. If we take a look at fault features presented on those figures for instance, we see that most of the faults impacting *Application Manager* Module are caused by other modules. On the other hand, *Teletext* Module has internal faults as much as faults stemming from the other modules. As such, distribution of features reveals characteristics of faults, errors and failures associated with individual elements of the architecture. This information is later utilized for architectural adjustment (See section 3.3.5).

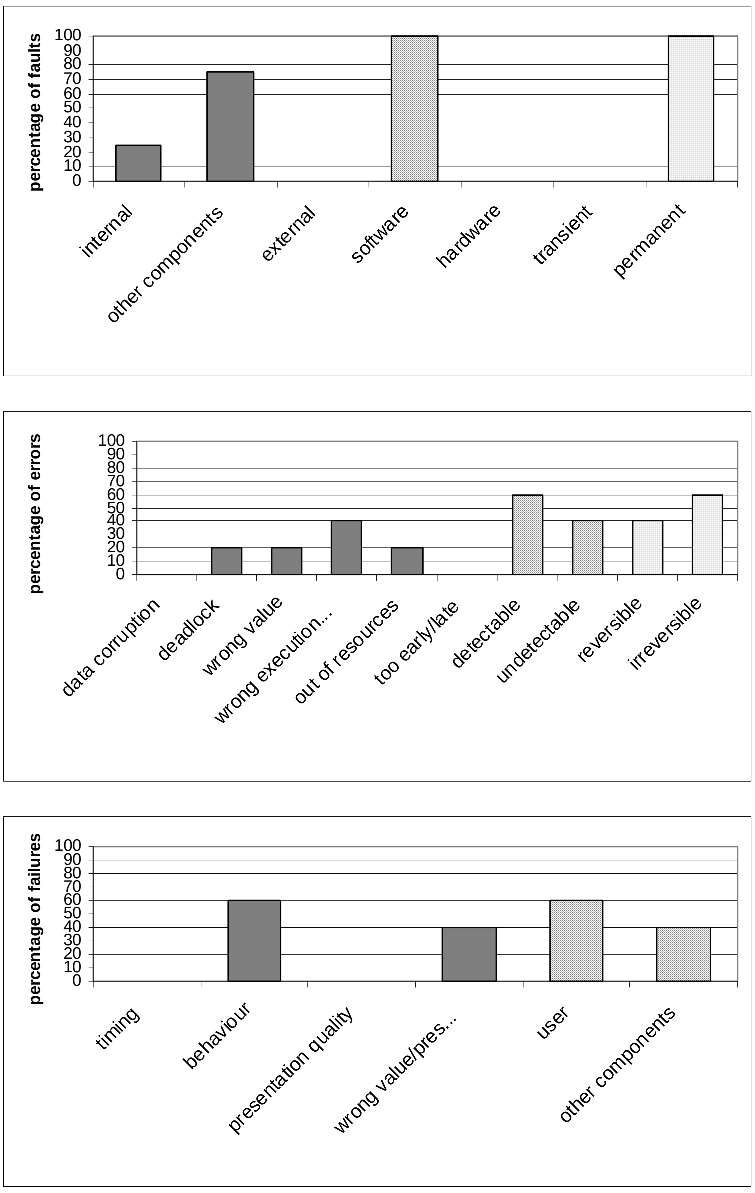


Figure 3.13: Fault, error and failure features associated with the *Application Manager* Module

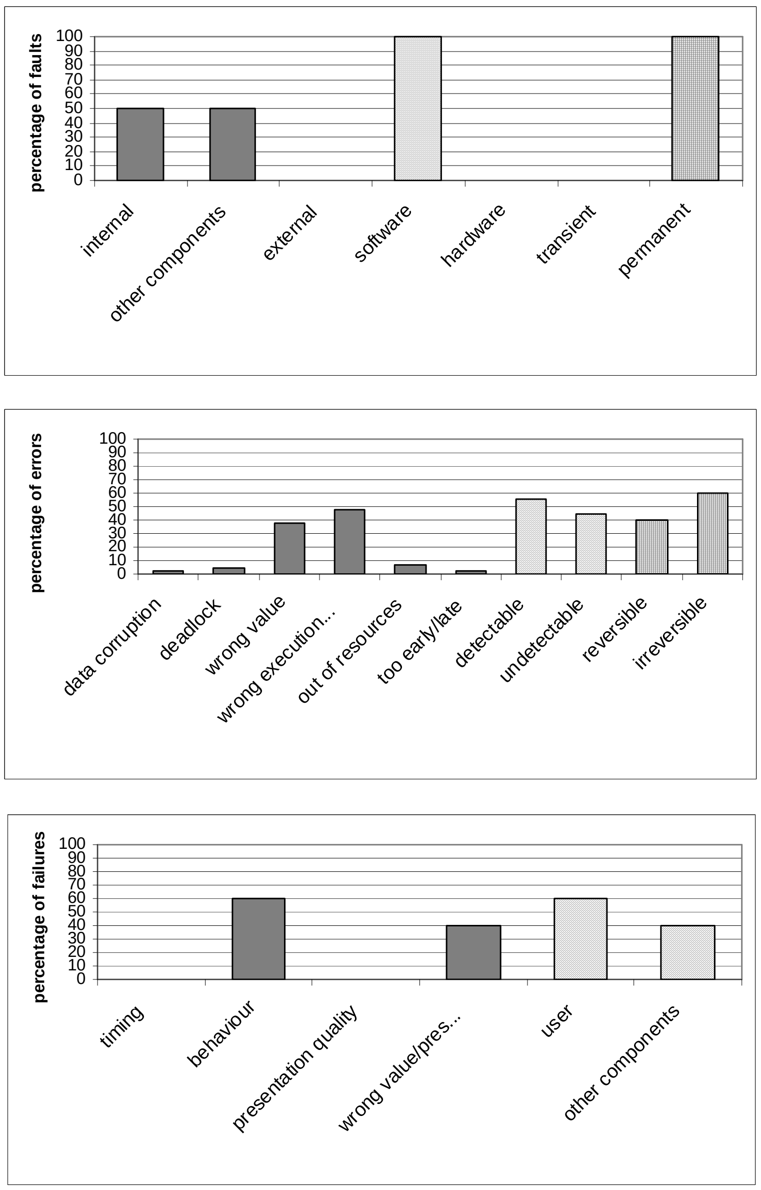


Figure 3.14: Fault, error and failure features associated with *Teletext* Module

Provide Failure Analysis Report

SARAH defines a detailed description of the fault tree sets, the failure scenarios, the architectural level analysis and the architectural element level analysis. These are described in the *failure analysis report* that summarizes the previous analysis results and provides hints for recovery. Sections comprised by the failure analysis report are listed in Table 3.4, which are in accordance with the steps of SARAH. The first section describes the project context, information sources and specific considerations (e.g. cost-effectiveness). The second section describes the software architecture. The third section presents the domain model of faults, errors and failures which include features of interest to the project. The fourth section contains list of failure scenarios annotated based on this domain model. The fifth section depicts the fault tree set generated from the failure scenarios together with the severity values assigned to each. The sixth and seventh sections include analysis results as presented in sections 3.3.4 and 3.3.4, respectively. Additionally, the sixth section includes the distribution of fault, error and failure features for all failure scenarios as depicted in Figure 3.15. Finally, the report includes a section on first hints for architectural recovery as titled *architectural tactics* [4]. This is explained in the next section.

Table 3.4: Sections of the failure analysis report.

1.	Introduction
2.	Software Architecture
3.	Failure Domain Model
4.	Failure Scenarios
5.	Fault Tree Set
6.	Architecture Level Analysis
7.	Architectural Element Level Analysis
8.	Architectural Tactics

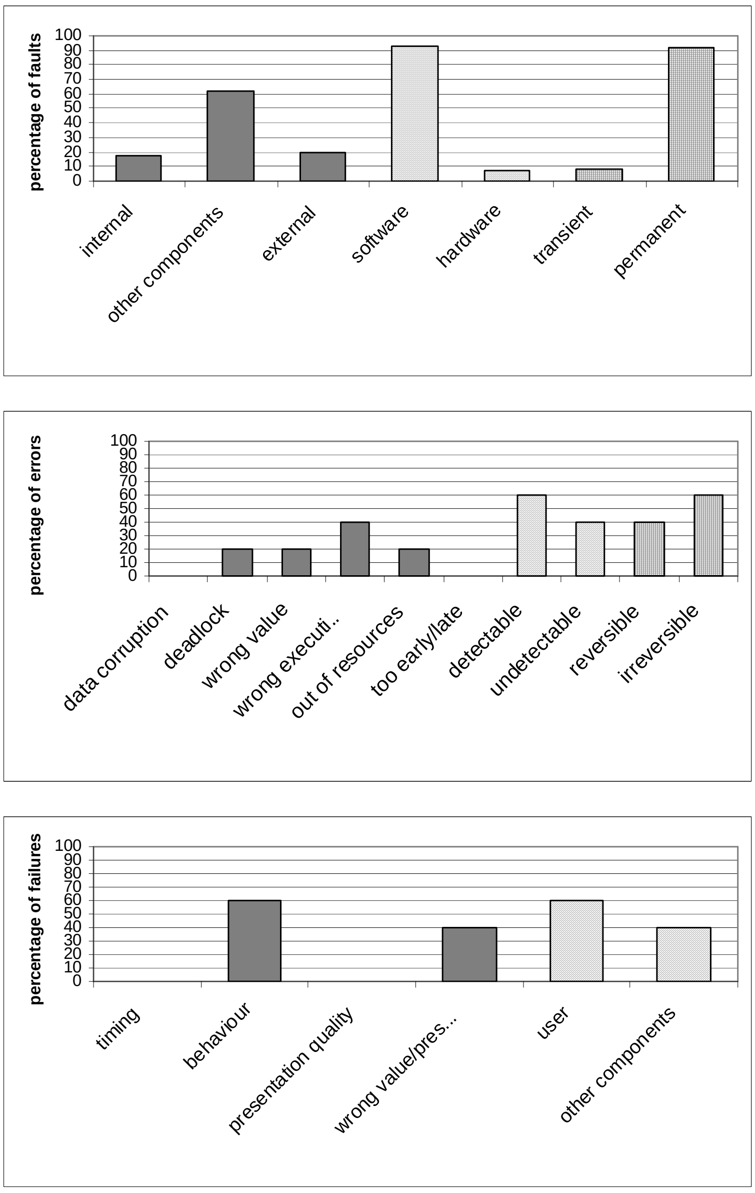


Figure 3.15: Feature distribution of fault, error and failures for all failure scenarios

3.3.5 Architectural Adjustment

The failure analysis report that is defined during the reliability analysis forms an important input to the architectural adjustment. Hereby the architecture will be enhanced to cope with the identified failures. This requires the following three steps: *i*) defining the elements to which the failure relates *ii*) identifying the architectural tactics and *iii*) application of the architectural tactics.

Table 3.5: Architectural element spots.

AEID	**Architectural Element Spot**
AMR	AC, AO, CB, CH, CMR, DDI, EPG, GC, LSM, PI, PM, TXT, VC, VO
AC	AMR, AP, LSM
AP	AC, DDI
AO	AMR, CA
CA	AMR, PM, T, AO, VO
CB	AMR, CMR
CH	AMR
CMR	AMR, CB
DDI	AMR, AP, CA, EPG, TXT, VP
EPG	AMR, DDI
G	GC
GC	AMR, G
LSM	AMR, AC, PM, VC
PI	AMR, PM
PM	AMR, CA, LSMR, T
T	CA, PM
TXT	AMR, DDI
VC	AMR, LSM, VP
VO	AMR, CA
VP	DDI, VC

Define Architectural Element Spots

Architectural tactics have been introduced as a characterization of architectural decisions that are used to support a desired quality attribute [4]. In SARAH we apply the concept of architectural tactics to derive architectural design decisions for supporting reliability. The previous steps in SARAH result in a prioritization of the most sensitive elements in the architecture together with the corresponding failures that might occur. Thus, SARAH prioritizes actually the design fragments [4] to which the specific tactics will be applied and to improve dependability. In general, for applying a recovery technique to an element we need also to consider the other elements coupled with it. This is because local treatments of individual elements might directly impact the dependent elements. For this reason, we define *architectural element spot* for a given element as the set of elements with which it interacts. This draws the boundaries of the design fragment [4] to which the design decisions will be applied. The coupling can be statically defined by analyzing the relations among the elements. Table 3.5 shows architectural element spots for each element in the example case. For example, Table 3.5 shows the elements that are in the architectural element spot of AMR as AC, AO, CB, CH, CMR, DDI, EPG, GC, LSM, PI, PM, TXT, VC and VO. These elements should be considered while incorporating mechanisms to AMR or any refactoring action that includes alteration of AMR and its interactions.

Identify Architectural Tactics

Once we have identified the sensitive elements, the element spots and the potential set of failure scenarios, we proceed with identifying the architectural tactics for enhancing dependability. As discussed in section 2.1.2, dependability means [3] are categorized as *fault forecasting*, *fault removal*, *fault prevention* and *fault tolerance* (See Figure 3.16). In SARAH, we particularly focus on fault tolerance techniques.

After identifying the sensitive elements we analyze the fault and error features related to these elements (Figures 3.13 and 3.14). Faults can be either internal or external to the sensitive element. If the source of the fault is internal this means that the fault tolerance techniques should be applied to the sensitive element itself. However, if the source of the fault is external, this means that the failures are caused by the other elements that are undependable. In that case, we might need to consider applying fault tolerance techniques to these undependable elements instead. These undependable elements can be traced with the help of the architectural element spot (section 3.3.5) and FTS. Some elements can tolerate external faults originating from other undependable elements. A broad set of fault tolerance techniques are available for this purpose such as exception handling, restarting, check-pointing and roll-back recovery [58]. On the other hand, if an element comprises faults that are *internal* and *permanent*, we might consider applying design diversity (e.g. recovery blocks, N-version programming [81]) on the element to tolerate such faults.

In Figure 3.16, some examples for fault tolerance techniques have been given. Each technique provides a solution for a particular set of problems. To make this explicit, each fault tolerance technique is tagged with error (and fault) types that it aims to

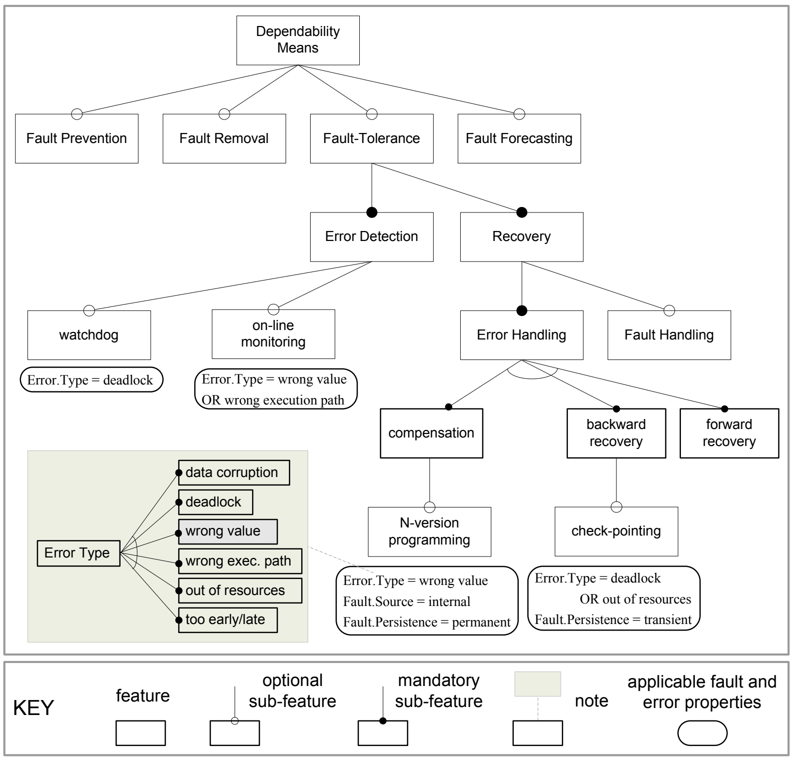


Figure 3.16: Partial categorization of dependability means with corresponding fault and error features

detect and/or recover. As an example, in Figure 3.16, *watchdog* [58] can be applied to detect *deadlock* errors for tolerating faults resulting in such errors. On the other hand, *N-version programming* [81] can be used for compensating wrong value errors that are caused by *internal* and *persistent* faults. Figure 3.16 shows only a few fault tolerance techniques as an example. Derivation of all fault tolerance tactics is out-of-scope of this work.

Apply Architectural Tactics

As the last step of architectural adjustment, selected architectural tactics should be applied to adjust the architecture if necessary.

The possible set of architectural tactics is determined as described in the previous section. Additionally, other criteria need to be considered including cost and limitations imposed by the system (resource constraints). For the given case, Table 3.6 shows, for example, the potential fault tolerance techniques that can be applied for AMR and TXT modules. The last column of the table shows the selected candidate

Table 3.6: Treatment approaches for sensitive components

AEID	Potential Means	Selected Means
AMR	on-line monitoring, watchdog, checkpoint & restart, resource checks, interface checks	on-line monitoring, interface checks
TXT	on-line monitoring, watchdog, n-version programming, checkpoint & restart, resource checks, interface checks	checkpoint & restart, resource checks

techniques.

Each architectural tactic has a set of characteristics, requirements, optional and alternative features and constraints. For instance, on-line monitoring is selected as a candidate error detection technique to be applied to AMR module. Schroeder defines in [108] basic concepts of on-line monitoring and explains its characteristics. Accordingly, a monitoring system is an external observer that monitors a fully functioning application. Based on the provided classification scheme [108] (purpose, sensors, event interpretation, action), we can specify an error detection mechanism as follows: The purpose of the mechanism is error detection. Sensors are the observers of a set of values represented in the element (i.e. AMR module) and they are always enabled. The event interpretation specification is built into the monitoring system. Sensing is based on sampling. Action specification is a recovery attempt, which can be achieved by partially or fully restarting the observed system and it is executed when a wrong value is detected.

The realization of an architectural tactic and measuring its properties (reengineering and performance overhead, availability, performability) for a system requires dedicated analysis and additional research. For example, in [74] software fault tolerance techniques that are based on diversity (e.g. recovery blocks, N-version programming) are evaluated with respect to dependability and cost. In chapter 5, we evaluate a local recovery technique and its impact on software decomposition with respect to performance and availability.

3.4 Discussion

The method and its application provide new insight in the scenario-based architecture analysis methods. A number of lessons can be learned from this study.

Early reliability analysis

Similar to other software architecture analysis methods, based on our experiences with SARAH we can state that early analysis of quality at the software architecture design level has a practical and important benefit [31]. In our case the early analysis relates to the analysis of the next release of a system (e.g. Digital TV) in a product line. Although, we did had access to the next release implementation of the Digital TV the reliability analysis with SARAH still provided useful insight in the critical failures and elements of the current architecture. For example, we were able to identify the critical modules *Application Manager*, *Teletext* and also got an insight in the failures, their causes and their effects. Without such an analysis it would be hard to denote the critical modules that have the risk to fail. For the next releases of the product this information can be used in deciding for the dependability means to be used in the architecture.

Utilizing a quality model for deriving scenarios

The use of an explicit domain model for failures has clearly several benefits. Actually, in the initial stages of the project we first tried to directly collect failure scenarios by interviewing several stakeholders. In our experience this has clear limitations because *i*) the set of failure scenarios for a given architecture is in theory too large and *ii*) even for the experts in the project it was hard to provide failure scenarios. To provide a more systematic and stable approach we have done a thorough analysis on failures and defined a fault domain model that represents essentially the reliability quality attribute. This model does not only provide systematic means for deriving failure scenarios but also defined the stopping criteria for defining failure scenarios. Basically we have looked at all the elements of the architecture, checked the failure domain and expressed the failure scenarios using the failure scenario template that we have presented in section 3.3.3. During the whole process we were supported by TV domain experts.

Impact of project requirements and constraints

From the industrial project perspective it was not sufficient to just define a failure domain model and derive the scenarios from it. A key requirement of the industrial case was to provide a reliability analysis that takes the user-perception as the primary criteria. This requirement had a direct impact on the way how we proceeded with the reliability analysis. In principle, it meant that all the failures that could be directly or indirectly perceived by the end-user had to be prioritized before the other failures. In our analysis this was realized by weighing the failures based on their severities from the user-perspective. In fact, from a broader sense, the focus on user-perceived failures could just be considered an example. SARAH provides a framework for reliability analysis and the method could be easily adapted to highlight other types of properties of failures.

Calculation of probability values of failures

One of the key issues in the reliability analysis is the definition of the probability values of the individual failures. In section 3.3.3 we have described three well-known strategies that can be adopted to define the probability values. In case more knowledge on probabilities is known in the project the analysis will be more accurate accordingly. As described in section 3.3.3 SARAH does not adopt a particular strategy and can be used with any of these strategies. We have illustrated the approach using fixed values.

Inherent dependence on domain knowledge

Obviously, the set of selected failure scenarios and values assigned to their attributes (severity) directly affect the results of the analysis. As it is the case with other scenario-based analysis methods both the failure scenario elicitation and the prioritization are dependent on subjective evaluations of the domain experts. To handle this inherent problem in a satisfactory manner SARAH guides the scenario elicitation by using the relevant failure domain model as described in section 3.3.3 and the use of failure scenario template in section 3.3.3. The initial assignment of severity values for the user-perceived failures is defined by the domain experts, but the independent calculation of the severities for intermediate failures is defined by the method itself as defined in section 3.3.3.

Extending the Fault Tree concept

In the reliability engineering community, fault trees have been used for a long time in order to calculate the probability that a system fails [34]. This information is derived from the probabilities of fault occurrences by processing the tree in a bottom-up manner. The system or system state can be represented by a single fault tree where the root node of the tree represents the failure of the system. In this context, failure means a crash-down in which no more functionality can be provided further. The fault tree can be also processed in a top-down manner to find the root causes of this failure. One of the contributions of this paper from the reliability engineering perspective is the Fault Tree Set (FTS) concept, which is basically a set of fault trees. The difference of FTS from a single fault tree is that an intermediate failure can participate in multiple fault trees and there exists multiple root nodes each of which represents a system failure perceived by the user. This refinement enables us to discriminate different types of system failures (e.g. based on severity) and infer what kind of system failures that an intermediate failure can lead to. Our analysis based on FTS is also different from the traditional usage of FTA. In FTA, the severity of the root node is not distributed to the intermediate nodes. Only the probability values are propagated to calculate the risk associated with the root node. In our analysis, we assign severity values to the intermediate nodes of FTS based on to what extend they influence the occurrence of root nodes and the severity of these nodes.

Architectural Tactics for fault tolerance

After the identification of the sensitive architectural elements, the related element spots and the corresponding failures in SARAH architectural tactics are defined. In the current literature on software architecture design this is not explicit and basically we had to rely on the techniques that are defined by the reliability engineering community. We have modeled a limited set of software fault tolerance techniques and defined the relation of these techniques with the faults and errors in the failure domain model that had been defined before. Bachman et al. derive architectural tactics by first listing concrete scenarios, matching these with general scenarios and deriving quality attribute frameworks [4]. In a sense this could be considered as a bottom-up approach. In SARAH the approach is more top-down because it essentially starts at the domain models of failures and fault tolerance techniques and then derives the appropriate matching. The concept of architectural tactics as an appropriate architectural design decision to enhance a particular quality attribute of the architecture remains of course the same.

3.5 Related Work

FMEA has been widely used in various industries such as automotive and aerospace. It has also been extended to make it applicable to the other domains or to achieve more analysis results. For instance, Failure Modes, Effects and Criticality Analysis (FMECA) is a well-known method that is built upon FMEA. As an extension to FMEA, FMECA [30] incorporates severity and probability assessments for faults. The probabilities of occurrence of faults (assumed to be known) are utilized together with the fault tree in order to calculate the probability that the system will fail. On the other hand, a severity is associated with every fault to distinguish them based on the cost of repair. Note that the severity definition in our method is different from the one that is employed by FMECA. We take a user-centric approach and define the severities for failures based on their impact on the user.

Almost all reliability analysis techniques have primarily devised to analyze failures in hardware components. These techniques have been extended and adapted to be used for the analysis of software systems. Application of FMEA to software has a long history [98]. Both FMEA and FTA have been employed for the analysis of software systems and named as Software Failure Modes and Effects Analysis (SFMEA) and Software Fault Tree Analysis (SFTA), respectively. In SFMEA, failure modes for software components are identified such as computational, logic and data I/O. This classification resembles the failure domain model of SARAH. However, SARAH separates fault, error and failure concepts and provides a more detailed categorization for each. Also, note that the failure domain model can vary depending on the project requirements and the system. In [76], SFTA is used for the safety verification of software. However, the analysis is applied at the source code level rather than the software architecture design level. Hereby, a set of failure-mode templates outlines the failure modes of programming language elements like assignments and conditional statements. These templates are composed according to the control flow of the program to derive a fault tree for the whole software.

In general, efforts for applying reliability analysis to software [62] mainly focus on the safety-critical systems, whose failure may have very serious consequences such as loss of human life and large-scale environmental damage. In our case, we focus on consumer electronics domain, where the systems are usually not safety-critical. FMEA has been used in other domains as well, where the methodology is adapted and extended accordingly. To use FMEA for analyzing the dependability of Web Services, new failure taxonomy, intrusions and various failure effects (data loss, financial loss, denial of service, etc.) are taken into account in [47]. Utilization of FMEA is also proposed in [128] for early robustness analysis of Web-based systems. The method is applied together with the Jacobson's method [100], which identifies three types of objects in a system: *i*) boundary objects that communicate with actors in a use-case, *ii*) entity objects that are objects from the domain and *iii*) control objects that serve as a glue between boundary objects and entity objects. In our method, we do not presume any specific decomposition of the software architecture and we do not categorize objects or components. However, we categorize failure scenarios based on the failure domain model and each failure scenario is associated with a component.

Jacobson's classification [100] is aligned with our failure domain model with respect to the propagation of failures (fault.source, failure.target). The target feature of failure, for instance, can be the user (i.e. actor) or the other components. In [124], a modular representation called Fault Propagation and Transformation Calculus (FPTC) is introduced. FPTC is used for specifying the failure behavior of each component (i.e. how a component introduces or transforms a failure type). This facilitates the automatic derivation of the propagation of the failures throughout the system. In our method, we represent the propagation of failure scenarios with fault trees. The semantics of the transformation is captured in the "type" tags of failure scenarios.

In this work, we made use of spreadsheets that define the failure scenarios and automatically calculate the severity values in the fault trees (after initial assignment of the user-perceived failure severities). This is a straightforward calculation and as such we have not elaborated on this issue. On the other hand, there is a body of work focusing on tool-support for FMEA and FTA. In [47], for example, FMEA tables are being integrated with a web service deployment architecture so that they can be dynamically updated by the system. In [90], fault trees are synthesized automatically. Further, multiple failures are taken into account, where a failure mode can contribute to more than one system failure. The result of the fault tree synthesis is a network of interconnected fault trees, which is analogous to the fault tree set in our method.

An Advanced Failure Modes and Effect Analysis (AFMEA) is introduced in [39], which also focuses on the analysis at the early stages of design. However, the aim of this work is to enhance FMEA in terms of the number and range of failure modes captured. This is achieved by constructing a behavior model for the system.

3.6 Conclusions

In this chapter, we have introduced the software architecture reliability analysis method (SARAH). SARAH utilizes and integrates mature reliability engineering techniques and scenario-based architectural analysis methods. The concept of fault, error failure models, the failure scenarios, fault tree set and the severity calculations are inspired from the reliability engineering domain [3]. The overall scenario-based elicitation and prioritization is derived from the work on software architecture analysis methods [31]. SARAH focuses on the reliability quality attribute and uses failure scenarios for analysis. Failure scenarios are prioritized based on the user perception. We have illustrated the steps of SARAH for analyzing reliability of a DTV software architecture. SARAH has helped us to identify the sensitivity points and architectural tactics for the enhancement of the architecture with respect to fault tolerance.

The basic limitation of the approach is the inherent subjectivity of the analysis results. SARAH provides systematic means to evaluate potential risks of the system and identify possible enhancements. However, the analysis results at the end depends on the failure scenarios that are defined. We use scenarios because they are practical and integrate good with the current methods. However, the correctness and completeness of these scenarios cannot be guaranteed in absolute sense because scenarios are subjective. Another critical issue is the definition of the probability values of failures in the analysis. However, just like existing reliability analysis approaches in the literature the accuracy of the analysis depends on the available knowledge in the project. SARAH does not adopt a fixed strategy but can be rather considered as a general process in which different strategies for defining probability values can be used.

Chapter 4

Software Architecture Recovery Style

The software architecture of a system is usually documented with multiple views. Each view helps to capture a particular aspect of the system and abstract away from the others. The current practice for representing architectural views focus mainly on functional properties of systems and they are limited to represent fault-tolerant aspects of a system. In this chapter, we introduce the *recovery style* and its specialization, the *local recovery style*, which are used for documenting a local recovery design.

The chapter is organized as follows. In section 4.1, we provide background information on documenting sofware architectures with multiple views. In section 4.2, we define the problem statement. Section 4.3 illustrates the problem with a case study, where we introduce local recovery to the open-source media player, MPlayer. This case study will be used in the following two chapters as well. In sections 4.4 and 4.5, we introduce the recovery style and the local recovery style, respectively. In section 4.6, we illustrate the usage of the recovery style for documenting recovery design alternatives of MPlayer software. We conclude this chapter after discussing alternatives, extensions and related work. The usage of the recovery style for supporting analysis and implementation are discussed in the following two chapters.

4.1 Software Architecture Views and Models

Due to the complexity of current software systems, it is not possible to develop a model of the architecture that captures all aspects of the system. A software architecture cannot be described in a simple one-dimensional fashion [18]. For this reason, as explained in section 2.2.1, architecture descriptions are organized around a set of *views*. An architectural view is a representation of a set of system elements and relations associated with them to support a particular concern [18]. Different views of the architecture support different goals and uses, often for different stakeholders.

In the literature initially some authors have prescribed a fixed set of views to document the architecture. For example, the Rational's Unified Process [70], which is based on Kruchten's 4+1 view approach [69] introduces the logical view, development view, process view and physical view. Another example is the Siemens Four Views model [56] that uses conceptual view, module view, execution view and code view to document the architecture.

Because of the different concerns that need to be addressed for different systems, the current trend recognizes that the set of views should not be fixed but different sets of views might be introduced instead. For this reason, the IEEE 1471 standard [78] does not commit to any particular set of views although it takes a multi-view approach for architectural description. IEEE 1471 indicates in an abstract sense that an architecture description consists of a set of views, each of which conforms to a *viewpoint* [78] associated with the various concerns of the stakeholders. Viewpoints basically represent the conventions for constructing and using a view [78] (See Figure 4.1).

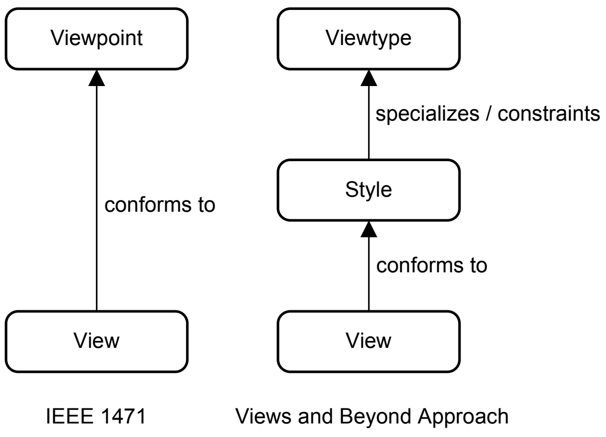


Figure 4.1: Viewpoint, view, viewtype and style

The Views and Beyond (V&B) approach as proposed by Clements et al. is another multi-view approach [18] that appears to be more specific with respect to the viewpoints. To bring order to the proposed views in the literature, the V&B approach holds that a system can be generally viewed from so-called three different *viewtypes*: *module viewtype*, *component & connector viewtype* and *allocation viewtype*. Each viewtype can be further specialized or constrained in so-called architectural *styles* (See Figure 4.1). For example, *layered style*, *pipe-and-filter style* and *deployment style* are specializations of the module viewtype, component & connector viewtype and allocation viewtype, respectively [18]. The notion of styles makes the V&B approach adaptable since the architect may in principle define any style needed.

4.2 The Need for a Quality-Based View

Certainly, existing multi-view approaches are important for representing the structure and functionality of the system and are necessary to document the architecture systematically. Yet, an analysis of the existing multi-view approaches reveals that they still appear to be incomplete when considering quality properties. The IEEE 1471 standard is intentionally not specific with respect to concerns that can be addressed by views. Thus, each quality property can be seen as a concern. In the V&B approach quality concerns appear to be implicit in the different views but no specific style has been proposed for this yet. One could argue that software architecture analysis approaches have been introduced for addressing quality properties. The difficulty here is that these approaches usually apply a separate quality model, such as queuing networks or process algebra, to analyze the quality properties. Although these models are useful to obtain precise calculations, they do not depict the decomposition of the architecture. They need to abstract away from the actual decomposition of the system to support the required analysis. As a complementary approach, preferably an architectural view is required to model the decomposition of the architecture based on the required quality concern.

One of the key objectives of the TRADER project [120] is to develop techniques for analyzing recovery at the architecture design level. Hereby, modules in a DTV can be decomposed in various ways and each alternative decomposition might lead to different recovery properties. To represent the functionality of the system we have developed different architectural views including module view, component & connector view and deployment view. None of these views however directly shows the decomposition of the architecture based on recovery concern. On the other hand, using separate quality models such as fault trees helps to provide a thorough analysis but is separate from the architectural decomposition. A practical and easy-to-use approach coherent with the existing multi-view approaches was required to understand the system from a recovery point of view.

In this chapter, we introduce the *recovery style* as an explicit style for depicting the structure of the architecture from a recovery point of view. The recovery style is a specialization of the module viewtype in the V&B approach. Similar to the module viewtype, component viewtype and allocation viewtype, which define units of decomposition of the architecture, the recovery style also provides a view of the architecture. Unlike quality models that are required by conventional analysis techniques, recovery views directly represent the decomposition of the architecture and as such help to understand the structure of the system related to the recovery concern. The recovery style considers *recoverable units* as first class elements, which represent the units of isolation, error containment and recovery control. Each recoverable unit comprises one or more modules. The dependency relation in module viewtype is refined to represent basic relations for coordination and application of recovery actions. As a further specialization of the recovery style, the *local recovery style* is provided, which is used for documenting a local recovery design in more detail. The usage of the recovery style is illustrated by introducing local recovery to the open-source software, MPlayer [85].

4.3 Case Study: MPlayer

We have applied local recovery to an open-source software, MPlayer [85]. MPlayer is a media player, which supports many input formats, codecs and output drivers. It embodies approximately 700K lines of code and it is available under the GNU General Public License. In our case study, we have used version v1.0rc1 of this software that is compiled on Linux Platform (Ubuntu version 7.04). Figure 4.2 presents a simplified module view [18] of the MPlayer software architecture with basic implementation units and direct dependencies among them. In the following, we briefly explain the important modules that are shown in this view.

- *Stream* reads the input media by bytes, or blocks and provides buffering, seek and skip functions.
- *Demuxer* demultiplexes (separates) the input to audio and video channels, and reads them from buffered packages.
- *Mplayer* connects the other modules, and maintains the synchronization of audio and video.
- *Libmpcodecs* embodies the set of available codecs.
- *Libvo* displays video frames.

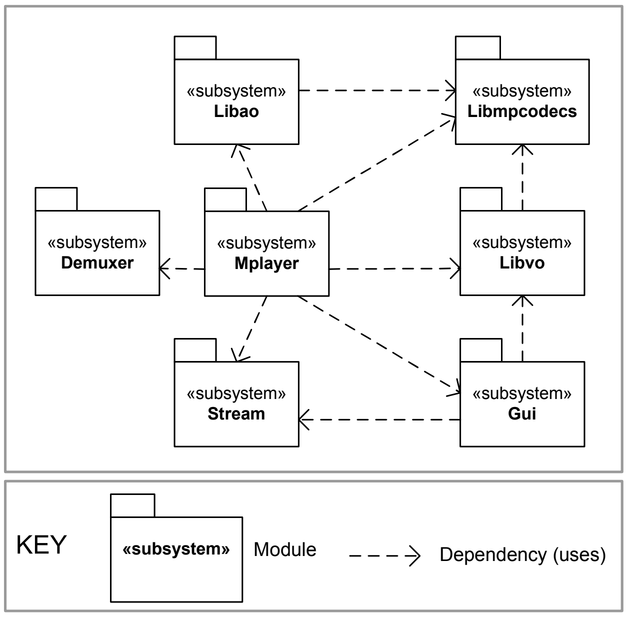


Figure 4.2: A Simplified Module View of the MPlayer Software Architecture

- *Libao* controls the playing of audio.
- *Gui* provides the graphical user interface of MPlayer.

We have derived the main modules of MPlayer from the package structure of its source code (e.g. Source files that are related to the graphical user interface are collected under the folder ./Gui).

4.3.1 Refactoring MPlayer for Recovery

In principle, fault tolerance should be considered early in the software development life cycle. Relevant fault classes, possible fault tolerance provisions should be carefully analyzed [95] and the software system must be structured accordingly [94].

However, there exist many software systems that have been already developed without fault tolerance in mind. Some of these systems comprise millions lines of code and it is not possible to develop these systems from scratch within time constraints. We have to refactor and possibly restructure them to incorporate fault tolerance mechanisms.

In general, software systems are composed of a set of modules. These modules provide a set of functionalities each of which has a different importance from the users' point of view [28]. A goal of fault tolerance mechanisms should be to make important functions available to the user as much as possible. For example, the audio/video streaming in a DTV should preferably not be hampered by a fault in a module that is not directly related to the streaming functionality.

Local recovery is an effective fault tolerance technique to attain high system availability, in which only the erroneous parts of the system are recovered. The other parts can remain available during recovery. As one of the requirements of local recovery, the system has to be decomposed into several units that are isolated from each other. Note that the system is already decomposed into a set of modules. Decomposition for local recovery is not necessarily, and in general will not be, aligned one-to-one with the module decomposition of the system. Multiple modules can be wrapped in a unit and recovered together. We have introduced a *local recovery* mechanism to MPlayer to make it fault-tolerant against transient faults. We had to decompose it into several units. The communication between multiple units had to be controlled and recovery actions had to be coordinated so that the units can be recovered independently and transparently, while the other units are operational.

4.3.2 Documenting the Recovery Design

Designing a system with recovery or refactoring it to introduce recovery requires the consideration of several design choices related to: the types of errors that will be recovered, error containment boundaries, the way that the information regarding error detection and diagnosis is conveyed, mechanisms for error detection, diagnosis, the control of the communication during recovery and the coordination of recovery actions, architectural elements that embody these mechanisms and the way that they interact with the rest of the system.

To be able to communicate these design choices, they should take part in the architecture description of the system. Effective communication is one of the fundamental purposes of an architectural description that comprise several views. However, viewpoints that capture functional aspects of the system are limited for representing recovery mechanisms and it is inappropriate for understandability to populate these views with elements and complex relations related to recovery.

Architecture description should also enable analysis of recovery design alternatives. For example, splitting the system for local recovery on one hand increases availability of the system during recovery but on the other hand, it leads to a performance overhead due to isolation. This leads to a trade-off analysis. Architecture description of the system should comprise information not only regarding the functional decomposition but also decomposition for recovery to support such analysis.

Realizing a recovery mechanism usually requires dedicated architectural elements and relations that have system-wide impact. Architectural description related to recovery is also required for providing a roadmap for the implementation and supporting the detailed design.

In short, we need a architectural view for recovery to *i*) communicate recovery design decisions *ii*) analyze design alternatives *iii*) support the detailed design. For this purpose, we introduce the recovery and the local recovery styles. In this chapter, we describe these styles and illustrate their usage for documenting the recovery design alternatives of MPlayer. In Chapter 5, we present analysis techniques that can be performed based on the recovery style. In Chapter 6, the style is used for supporting the detailed design and realization.

4.4 Recovery Style

The key elements and relations of the recovery style are listed below. A visual notation based on UML [104] is shown in Figure 4.3 that can be used for documenting a *recovery view*.

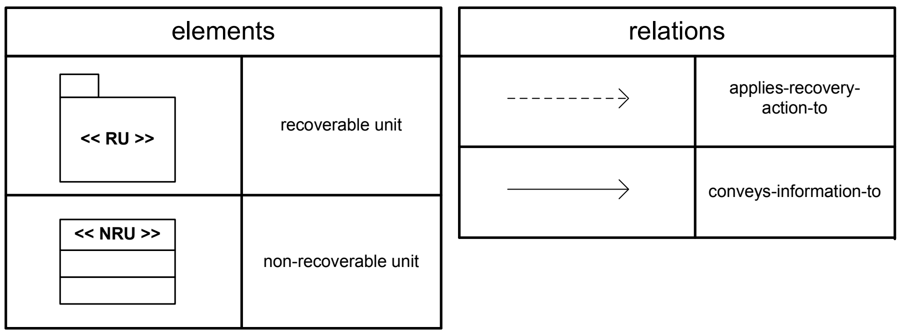


Figure 4.3: Recovery style notation

- *Elements*: recoverable unit (RU), non-recoverable unit (NRU)
- *Relations*: applies-recovery-action-to, conveys-information-to
- *Properties of elements*: Properties of RU: set of system modules together with their criticality (i.e. functional importance) and reliability, types of errors that can be detected, supported recovery actions, type of isolation (process, exception handling, etc.). Properties of NRU: types of errors that can be detected (i.e. monitoring capabilities).

- *Properties of relations*: Type of communication (synchronous / asynchronous) and timing constraints if there are any.
- *Topology*: The target of a *applies-recovery-action-to* relation can only be a *RU*.

The recovery style introduces two types of elements; *RU* and *NRU*. A RU is a unit of recovery in the system, which wraps a set of modules and isolates them from the rest of the system. It provides interfaces for conveying information about detected errors and responding to triggered recovery actions. A RU has the ability to recover independently from other RU and NRUs it is connected to. A NRU, on the other hand, can not be recovered independent from other RU and NRUs. It can only be recovered together with all other connected elements. This can happen in the case of global recovery or a recovery mechanism at a higher level in the case of a hierarchic decomposition.

The relation *conveys-information-to* is about communication between elements for guiding and coordinating the recovery actions. Conveyed information can be related to detected errors, diagnosis results or a chosen recovery strategy. The relation *applies-recovery-action-to* is about the control imposed to a RU and it affects the functioning of the target RU (suspend, kill, restart, roll-back, etc.)

4.5 Local Recovery Style

In the following, we present the local recovery style in which the elements and relations of the recovery style are further specialized. This specialization is introduced in particular to document the application of our local recovery framework (FLORA) to MPlayer as presented in Chapter 6. Figure 4.4 provides a notation for specialized elements and relations based on UML.

- *Elements*: RU, recovery manager, communication manager
- *Relations*: restarts, kills, notifies-error-to, provides-diagnosis-to, sends-queued message-to
- *Properties of elements*: Each RU has a *name* property in addition to those defined in the recovery style.
- *Properties of relations*: *notifies-error-to* relation has a *type* property to indicate the error type if multiple types of errors can be detected.

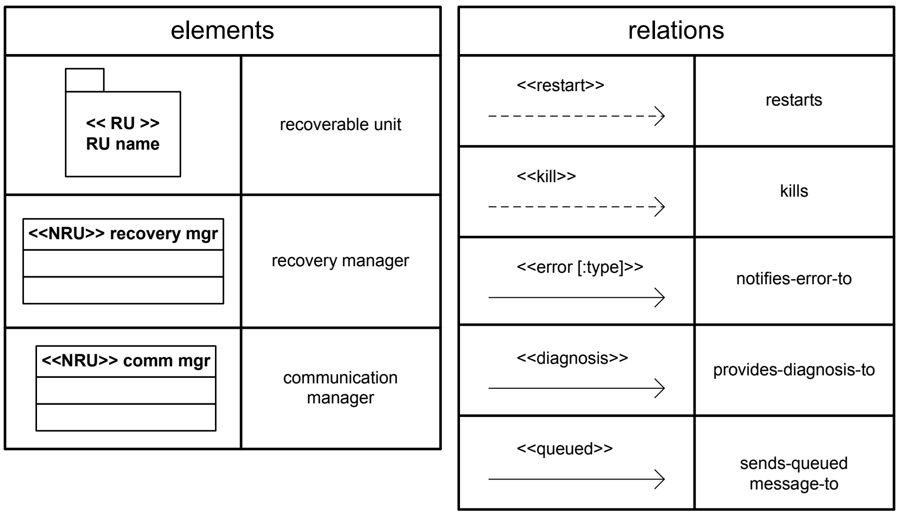


Figure 4.4: Local recovery style notation

- *Topology*: *restarts* and *kills* relations can be only from a *recovery manager* to a RU. *sends-queued message-to* can be between a RU and a communication manager.

Communication manager connects RUs together, routes messages and informs connected elements about the availability of another connected element. *Recovery manager* applies recovery actions on RUs.

Figure 4.5 depicts a simple instance of the local recovery style with one communication manager, one recovery manager and two RUs, A and B. Errors are detected by the communication manager and notified to the recovery manager. Recovery manager restarts A and/or B. The messages that are sent from the communication manager to A are stored in a queue by the communication manager while A is unavailable (i.e. being restarted) and sent (retried) after A becomes available.

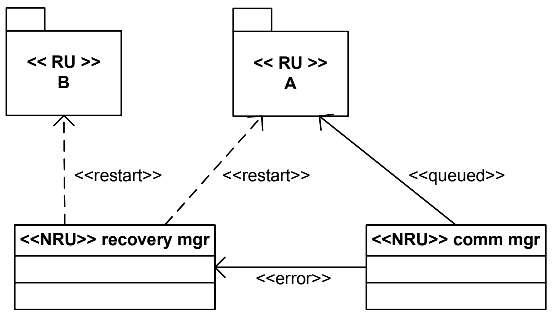


Figure 4.5: A simple recovery view based on the local recovery style (KEY: Figure 4.4)

4.6 Using the Recovery Style

Our aim is to use the recovery style for MPlayer (Section 4.3) to choose and realize a recovery design among several design alternatives. One such alternative is to introduce a local recovery mechanism comprising 3 RUs as listed below.

- *RU AUDIO*: provides the functionality of *Libao*
- *RU GUI*: encapsulates the *Gui* functionality
- *RU MPCORE*: comprises the rest of the system.

Figure 4.6 depicts the boundaries of these RUs, which are overlayed on the module view of the MPlayer software architecture. In Figure 4.7(b) the recovery design corresponding to this RU selection is shown[1]. Here, we can also see two new architectural elements that are not recoverable; a *communication manager* that mediates and controls the communication between the RUs and a *recovery manager* that applies the recovery actions on RUs.

Note that Figure 4.7(b) shows just one possible design for the recovery mechanism to be introduced to the MPlayer. We could consider many other design alternatives. One alternative would be to have only 1 RU (See Figure 4.7(a)). This would basically lead to a global recovery since all the modules are encapsulated in a single RU. We could also have more than 3 recoverable units as shown in Figure 4.7(c).

[1]In this chapter, we use the recovery style notation to represent all the recovery design alternatives. We use the local recovery style in Chapter 6, to represent local recovery designs in more detail.

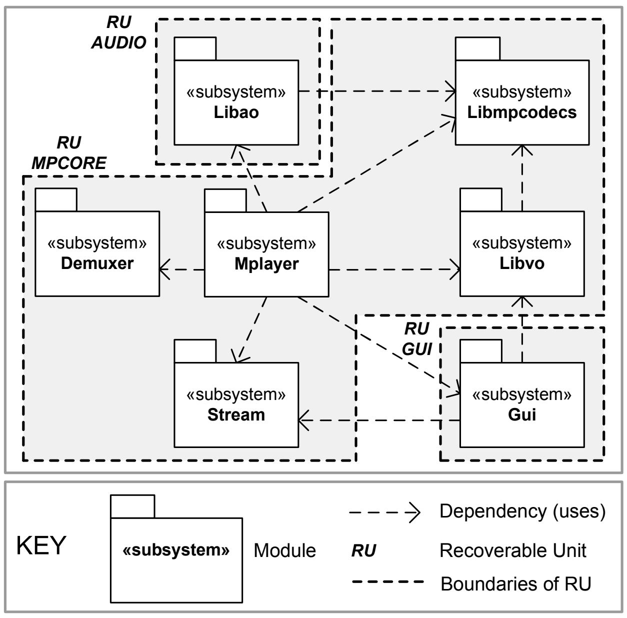
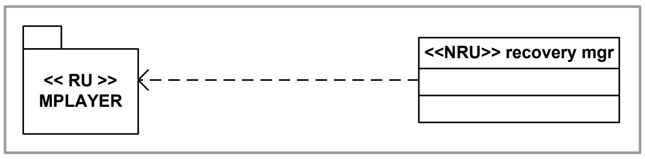


Figure 4.6: The Module View of the MPlayer Software Architecture with the Boundaries of the Recoverable Units

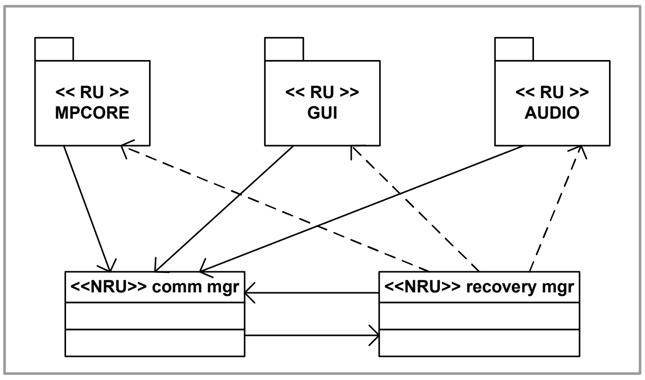
Hereby, each module of the system is assigned to a separate RU[2]. We could consider many such alternatives to decompose the system into a set of RUs[3]. On the other hand, there could be more than one element that controls the recovery actions and communication (e.g. distributed or hierarchical control). These elements could be recoverable units as well. Recovery views of the system helps us to document and communicate such design alternatives. The following chapters further elaborate on the related analysis techniques and realization activities supported by the recovery style.

[2]The names of the RUs are not significant and they are assigned based on the comprised modules.

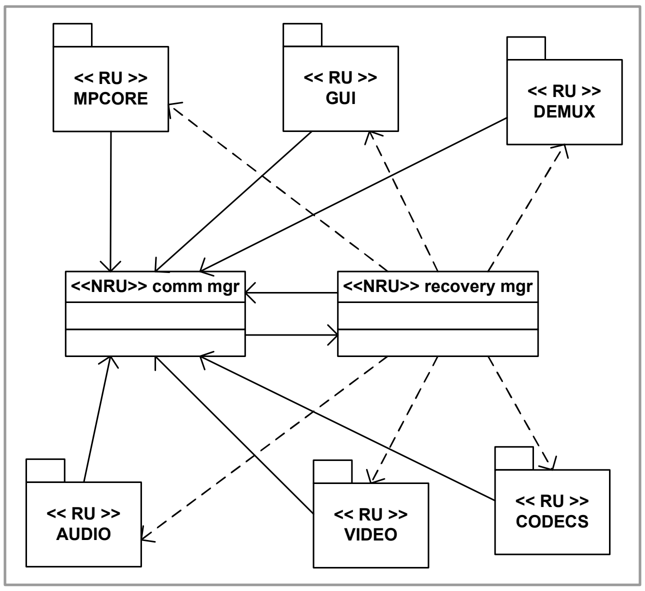
[3]Chapter 5 presents an analysis method together with a tool-set for optimizing the decomposition of software architecture into RUs.



(a) recovery design with 1 RU



(b) recovery design with 3 RUs



(c) recovery design with 7 RUs

Figure 4.7: Architectural alternatives for the recovery design of MPlayer software (KEY: Figure 4.3)

4.7 Discussion

Style or viewpoint?

In this work we have extended the module viewtype to represent a recovery style. In the V&B approach a distinction is made between viewtypes, styles and views [18]. Hereby, styles can be considered as specialized viewtypes, and views are considered as instances of these styles. The IEEE 1471 standard [78], on the other hand, describes viewpoints and views, and does not describe the concept of style. In the IEEE 1471 standard, viewpoint represents the language to represent views. So, in the parlance of this standard, the recovery style that we have introduced can be considered as a viewpoint as well because it represents a language to define recovery views. Yet, in this paper we have focused on representing recovery views for units of implementation which is made explicit in the V&B approach through the module viewtype. To clarify this focus we adopted the term style instead of the more general term viewpoint.

Recovery Style based on other viewtypes

Although we have focused on defining a recovery view for implementation units, we could also derive a recovery style for the component & connector viewtype or the allocation viewtype [18] to represent recovery views of run-time or deployment elements. This would provide a broader view on recovery and we consider this as a possible extension of our work.

Utilization of Recovery Style in different contexts

There are several works that apply local recovery in different contexts. They can be represented with the local recovery style as well. For example, in [53] device drivers are executed on separate processes, where microreboot [16] is applied to increase the failure resilience of operating systems. They provide a view of the architecture of the operating system, where the architectural elements related to fault tolerance and corresponding relations (recovery actions and error notifications) are shown. According to this view [53] *Process Manager*, which restarts the processes, is a non-recoverable unit that coordinates the recovery actions. *Reincarnation Server* monitors the system to facilitate error detection and diagnosis and it guides the recovery procedure. *Data Store*, which is a name server for interconnected components, mediates and controls communication.

4.8 Related Work

Perspectives [125] guide an architect to modify a set of existing views to document and analyze quality properties. Our work addresses the same problem for recovery by introducing the recovery style, which is instantiated as a view. One of the motivations for using perspectives instead of creating a quality-based view is to avoid the duplicate information related to architectural elements and relations. This was less an issue in our project context, where we needed an explicit view for recovery to depict the related decomposition, which includes, dedicated architectural elements with complex interactions among them (e.g. Figure 6.3).

Architectural tactics [4] aim at identifying architectural decisions related to a quality attribute requirement and composing these into an architecture design. The recovery style is used for representing and analyzing an architectural tactic for recovery. It provides a view of the system, by means of which the recovery design of a system can be captured.

The idealized fault-tolerant COTS (commercial off-the-shelf) component is introduced for encapsulating COTS components to add fault tolerance capabilities [25]. The approach is based on the idealized C2 component (iC2C), which is compliant with the C2 architectural style [114]. The C2 architectural style is a component-based style, in which components communicate only through asynchronous messages mediated by connectors. Components are wrapped to cope with interface and architectural mismatches [43]. iC2C specializes the elements and relations of the C2 architectural style to explicitly represent fault tolerance provisions. The iC2C differentiates between service requests/responses and interface/failure exceptions as subtypes of requests and notifications in the C2 architectural style. It also introduces two internal components, a *NormalActivity* component and an *AbnormalActivity* component together with an additional connector between these. The iC2C *NormalActivity* component implements the normal behavior and error detection, whereas the iC2C *AbnormalActivity* component is responsible for error recovery.

The idealized fault-tolerant architectural element (iFTE) [27] is introduced to make exception handling capabilities explicit at the architectural level. The iFTE adopts the basic principles of the iC2C. However, it can be instantiated for both components and connectors, named as *iFTComponent* and *iFTConnector*, respectively. This approach enables the analysis of recovery properties of a system based on its architecture design.

Multi-version connector [93] is a type of connector based on the C2 architectural style. It relies on multiple versions of a component for different operations to improve the dependability of a system. This approach can be used for representing N-version programming in the software architecture design.

The notion of Web Service Composition Action (WSCA) [113] is introduced for structuring the composition of Web services in terms of coordinated atomic actions to enforce transaction processing. An XML-based specification defines the elements, which participate to perform the corresponding action, together with the standard behavior and their cooperative recovery actions in case of an exception.

An architectural style for tolerating faults in Web services is proposed in [91]. In this style, a system is composed of multiple variants of components, which crash in case of a failure. The system is then dynamically reconfigured to utilize alternative variants of the crashed components.

In the architecture-based exception handling [63] approach, architecture configuration exceptions and their handling are specified in the architecture description. Configuration exceptions are defined as conditions with respect to possible configurations of architectural elements and their interaction (communication patterns within the current configuration). Exception handlers are defined as a set of required changes to the architecture configuration (substitution of elements and alterations of bindings). The approach is supported by a run-time environment, which reconfigures the system dynamically based on the provided specification.

Another run-time architecture reconfiguration approach is presented in [24] for updating a running system to tolerate faults. In this approach, the architecture description is expressed with xADL [23] and repair plans are specified as a set of elements to be added to or removed from the architecture. In case of an error, the associated repair plan is executed and the configuration of the architecture is changed accordingly.

A generic run-time adaptation framework is presented in [44] together with its specialization and realization for performance adaptation. According to the concepts in this framework, our *monitoring mechanisms* are error detectors. Diagnosis facilities can be considered as *gauges*, which interpret monitored low-level events, and recovery actions can be considered as *architectural repairs*. As a difference from the approach presented in [44], we propose a style specific for fault tolerance.

As previously discussed in Chapter 3, several software architecture analysis approaches have been introduced for addressing quality properties. The goal of these approaches is to assess whether or not a given architecture design satisfies desired quality requirements. The main aim of the recovery style, on the other hand, is to communicate and support the design of recovery. Analysis is a complementary work to make trade-off decisions, tune and select recovery design alternatives.

4.9 Conclusions

In this chapter, we have introduced the recovery style to document and analyze recovery properties of a software architecture. The recovery style is a specialization of the module viewtype as defined in the V&B approach [18]. We have further specialized the recovery style to define the local recovery style. The usage of these styles have been illustrated for defining recovery views of MPlayer. These views have formed the basis for analysis of design alternatives. They have also supported the detailed design and implementation of local recovery for MPlayer. Chapter 5 explains the analysis of decomposition alternatives for local recovery, whereas Chapter 6 explains the realization of recovery views.

Chapter 5

Quantitative Analysis and Optimization of Software Architecture Decomposition for Recovery

Local recovery enables the recovery of the erroneous parts of a system while the other parts of the system are operational. To prevent the propagation of errors, the architecture needs to be decomposed into several parts that are isolated from each other. There exist many alternatives to decompose a software architecture for local recovery. It appears that each of these decomposition alternatives performs differently with respect to availability and performance metrics. In this chapter, we propose a systematic approach, which employs integrated set of analysis techniques to optimize the decomposition of software architecture for local recovery.

The chapter is organized as follows. In the following two sections, we introduce background information on program analysis and analytical models that are utilized by the proposed approach. In section 5.3, we define the problem statement for selecting feasible decomposition alternatives. In section 5.4, we present the top-level process of our approach and the architecture of the tool that supports this process. Sections 5.5 through 5.11 explain the steps of the top-level process and illustrate them for the open-source MPlayer software. In section 5.12 we evaluate the analysis results. We conclude the chapter after discussing limitations, extensions and related work.

5.1 Source Code and Run-time Analysis

Program analysis techniques are used for automatically analyzing the structure and/or behavior of software for various goals such as the ones listed below.

- Optimizing the generated code during compilation [1]
- Automatically pinpointing (potential) errors and as such reducing debugging time [119]
- Automatically detecting vulnerabilities and security threats [92]
- Supporting the understanding and comprehension of large and complex programs [20]
- Reverse engineering legacy software systems [87]

Several types of program analysis techniques are used in practice as a part of existing software development tools and compilers, which are tailored for different execution platforms and programming languages. There are mainly two approaches for program analysis; *static analysis* and *dynamic analysis*.

Static analysis approaches inspect the source code of programs and perform the analysis without actually executing these programs. The sophistication of the analysis varies depending on the granularity of the analysis (e.g. individual statements, functions/methods, source files) and the purpose (e.g. spotting potential errors, verifying specified properties) [9]. Although static analysis approaches are helpful for several aforementioned purposes, they are limited due to the lack of run-time information. For example, static analysis can reveal the dependency between function calls but not the frequency of calls and their execution times. Moreover, static analysis is not always practical and scalable. Depending on the type of analysis (e.g. data flow and especially pointer analysis [55]) and the size of the analyzed program, static analysis can become highly resource consuming, and very soon intractable.

Dynamic analysis approaches make use of data gathered from a running program. The program is executed on a real or virtual processor according to a certain scenario. During the execution, various data are collected that are subject to analysis. The advantage of dynamic analysis approaches is their ability to capture detailed interactions at run-time (e.g. pointer operations, late binding). This leads to more accurate information about the analyzed program compared to what might be obtained with static analysis. On the other hand, dynamic analysis has also limitations and drawbacks. First, the collected data for analysis depends on the program input.

Use of techniques for estimating *code coverage* [127] can help to ensure that a sufficient subset of the program's possible behaviors has taken into account. Second, depending on the type and granularity of the analyzed information, the amount of collected data may turn out to be huge and intractable. Third, instrumentation or probing has an effect on the execution of the target program. As a potential risk, this effect can influence the program behavior and analysis results.

In this chapter, we utilize dynamic analysis techniques for estimating the performance overhead introduced by a recovery design. We utilize two different tools; *GNU gprof* [40] and *Valgrind* [88]. We use *GNU gprof* tool to obtain the function call graph of the system together with statistics about the frequency of performed function calls and the execution time of functions (i.e. function call profile). We use the *Valgrind* framework to obtain the data access profile of the system.

5.2 Analysis based on Analytical Models

The main goal of recovery is to achieve higher system *availability* to its users in case of component failures. Thus, to select an optimal recovery design, we need to quantitatively assess design alternatives with respect to availability. As defined in section 2.1.1, availability is a measure of readiness for correct service and it can be simply represented by the following formula.

$$Availability = \frac{MTTF}{MTTF + MTTR} \tag{5.1}$$

Hereby, $MTTF$ and $MTTR$ stand for the *mean time to failure* and the *mean time to repair*, respectively. These concepts are mainly inherited from hardware fault tolerance and adapted for software fault tolerance. To estimate the availability and related quality attributes for complex systems and fault-tolerant designs, several formal models, techniques and tools have been devised [121]. These techniques are mostly based on Markov models [51], queuing networks [22], fault trees [35], or a combination of these. Quantitative availability analysis methods usually employ so-called *state-based* models [61]. In the application of state-based models, it is in general assumed that the behavior of modules has a Markov property, where the future behavior is conditionally independent of the past behavior [48]. Usually *discrete time Markov chains (DTMC)* or *continuous time Markov chains (CTMC)* are used as the modeling formalism. DTMCs are used for representing applications that operate on demand, whereas CTMC formalism is well-suited for continuously operating software applications [48]. A CTMC is defined by a set of states and transitions between these states. The transitions are labeled with rates λ indicating that the transition is taken after a delay that is governed by an exponential distribution with parameter λ. In availability modeling, the states represent the degree of availability of the system [73]. Figure 5.1 depicts a CTMC as a simple availability model.

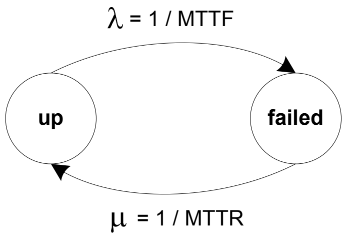


Figure 5.1: An example CTMC as a simple availability model

Since the delay for taking a transition is governed by an exponential distribution with a constant rate, the mean time to take a transition is simply the reciprocal of the transition rate. Consequently, $\lambda = 1/MTTF$ defines the failure rate, where a transition is made from the "up" state to the "failed" state. Similarly, $\mu = 1/MTTR$ defines the recovery rate, where a transition is made from an "failed" state to the "up" state.

Markov model checker tools like CADP [42] can take a CTMC model as an input and compute the probability that a particular state will be visited in the long run (i.e. steady state). Since states represent availability degrees, this makes it possible to calculate the total availability of the system in the long run. In the example model shown in Figure 5.1, for instance, the availability degrees implied by the "up" and "failed" states are 1 and 0, respectively. Thus, the probability that the "up" state is visited in the long run determines the availability of the system.

A potential problem with such analytical models is the *state space explosion* problem. Depending on the complexity of the system and the recovery design, the state space can get too large. As a result, it can take too long time for the model checker to calculate, if possible at all, the steady state probabilities. To overcome this problem, *Input/output interactive Markov chain (I/O-IMC)* formalism has been introduced [10] as explained in the following subsection.

5.2.1 The I/O-IMC formalism

I/O-IMCs [10, 11] are a combination of Input/Output automata (I/O automata) [86] and interactive Markov chains (IMCs) [54]. Essentially, I/O-IMCsare extensions of CTMCs [57] having 2 main types of transitions; *interactive transitions* and *Markovian transitions*. Markovian transitions are labeled with the corresponding transition rates just like in regular CTMCs. Interactive transitions are identified with 3 types of actions;

- *Input actions*: This type of actions are controlled by the environment. A transition with an input action is taken if and only if another I/O-IMC performs the corresponding output action (input and output actions are matched by labels). I/O-IMCs are *input-enabled* meaning that each state is ready to respond to any input action. Hence, every state has an outgoing transition (or self-transition by default), for each possible input action.
- *Output actions*: These actions are controlled by the I/O-IMC itself. Transitions labeled with output actions are taken immediately.
- *Internal actions*: These actions are not visible to the environment. They are controlled by the I/O-IMC itself and the corresponding transitions are taken immediately.

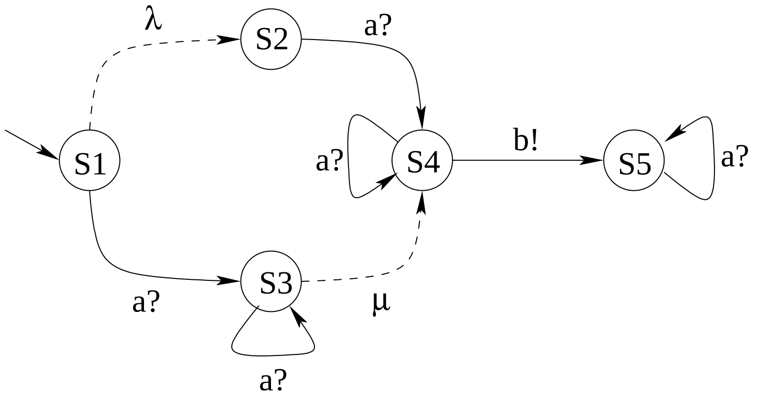


Figure 5.2: An example I/O-IMC

An example I/O-IMC is depicted in Figure 5.2. Hereby, Markovian transitions are depicted with dashed lines, and interactive transitions by solid lines. Labels of input and output actions are appended with "?" and "!", respectively. In this example, there are 2 Markovian transitions: one from $S1$ to $S2$ with rate λ and another from $S3$ to $S4$ with rate μ. The I/O-IMC has 1 input action, which is labeled as $a?$. Note that, all the states have a transition for this input action to ensure that the I/O-IMC is *input-enabled*. There is also only 1 output action specified in this example, which is labeled as $b!$. $S1$ is the initial state of the I/O-IMC.

The behavior of an I/O-IMC depends on both its internally specified behavior and the environment (i.e. behavior of the other I/O-IMCs). For example, the state $S1$ in Figure 5.2 has 2 outgoing transitions: a Markovian transition to $S2$ with rate λ and an interactive transition to $S3$ with input action $a?$. In $S1$, the I/O-IMC delays for a time that is governed by an exponential distribution with parameter λ, and moves to $S2$. If however, before that delay ends, an input $a?$ arrives (another I/O-IMC takes a transition with output action $a!$), then the I/O-IMC moves to $S3$. As soon as the state $S4$ is reached, the I/O-IMC takes a transition to $S5$ with output action $b!$. This transition might influence the behavior of other I/O-IMCs that have been waiting for an input action $b?$.

The I/O-IMC formalism enables modularity in model building and analysis. Multiple I/O-IMC models can be composed together based on the common inputs they consume and outputs they generate. Instead of building a model for the system as a whole, the behavior of system elements can be modeled independently and then combined with a *compositional aggregation approach* [10]. Compositional aggregation is a technique to build an I/O-IMC by composing, in successive iterations, a number of elementary I/O-IMCs and reducing (i.e. aggregating) the state-space of the generated I/O-IMC after each iteration. This helps in managing large state spaces and model elements can be re-used in different configurations. Composition and aggregation of I/O-IMCs results in a regular CTMC, which can be then analyzed using standard methods to compute different dependability and performance measures.

5.3 Software Architecture Decomposition for Local Recovery

An error occurring in one part of the system can propagate and lead to errors in other parts. To prevent the propagation of errors and recover from them locally, the system should be decomposed into a set of isolated *recoverable units* (RUs) ([59, 16]). For example, recall the MPlayer case study presented in section 4.3. As re-depicted in Figure 5.3, one possible decomposition for the MPlayer software is to partition the system modules into 3 RUs; *i*) *RU AUDIO*, which provides the functionality of *Libao* *ii*) *RU GUI*, which encapsulates the *Gui* functionality and *iii*) *RU MPCORE* which comprises the rest of the system. This is only one alternative decomposition for isolating the modules within RUs and as such introducing local recovery. Obviously, the partitioning can be done in many different ways.

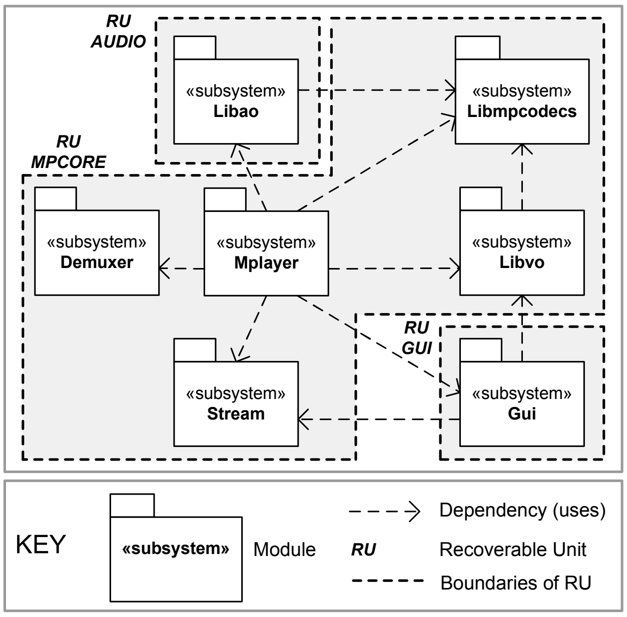


Figure 5.3: An example decomposition of the MPlayer software into RUs

5.3.1 Design Space

To reason about the number of decomposition alternatives, we first need to model the design space. In fact, the partitioning of architecture into a set of RUs can be generalized to the well-known *set partitioning problem* [50]. Hereby, a *partition* of a set S is a collection of disjoint subsets of S whose union is S. For example, there exists 5 alternatives to partition the set $\{a, b, c\}$. These alternatives are: $\{\{a\}, \{b\}, \{c\}\}$, $\{\{a\}, \{b, c\}\}$, $\{\{b\}, \{a, c\}\}$, $\{\{c\}, \{a, b\}\}$, $\{\{a, b, c\}\}$. The number of ways to partition a set of n elements into k nonempty subsets is computed by the so-called *Stirling numbers of the second kind*, $S(n, k)$ [50]. It is calculated with the recursive formula shown in Equation 5.2.

$$\left\{ {n \atop k} \right\} = k \left\{ {n-1 \atop k} \right\} + \left\{ {n-1 \atop k-1} \right\}, n \geq 1 \tag{5.2}$$

The total number of ways to partition a set of n elements into arbitrary number of nonempty sets is counted by the n^{th} *Bell number* as follows.

$$B_n = \sum_{k=1}^{n} S(n,k) \tag{5.3}$$

In theory, B_n is the total number of partitions of a system with n modules. B_n grows exponentially with n. For example, $B_1 = 1$, $B_3 = 5$, $B_4 = 15$, $B_5 = 52$, $B_7 = 877$ (The MPlayer case), $B_{15} = 1.382.958.545$. Therefore, searching for a feasible design alternative becomes quickly problematic as n (i.e. the number of modules) grows.

5.3.2 Criteria for Selecting Decomposition Alternatives

Each decomposition alternative in the large design space may have both pros and cons. In the following, we outline the basic criteria that we consider for choosing a decomposition alternative.

- *Availability*: Local recovery aims at keeping the system available as much as possible. The total availability of a system depends on the availability of its individual recoverable units. To maximize the availability of the overall system, $MTTF$ of RUs must be kept high and $MTTR$ of RUs must be kept low (See Equation 5.1). The $MTTF$ and $MTTR$ of RUs on their turn depend on the $MTTF$ and $MTTR$ values of the contained modules. As such, the overall value of $MTTF$ and $MTTR$ properties of the system depend on the decomposition alternative, that is, how we separate and isolate the modules into a set of RUs.

- *Performance*: Logically, the optimal availability of the system is defined by the decomposition alternative in which all the modules in the system are separately isolated into RUs. However, increasing the number of RUs leads to a performance overhead due to the dependencies between the separated modules in different RUs. We distinguish between two important types of dependencies that cause a performance overhead; *i*) *function dependency* and *ii*) *data dependency*.

  The function dependency is the number of function calls between modules across different RUs. For transparent recovery these function calls must be redirected, which leads to an additional performance overhead. For this reason, for selecting a decomposition alternative we should consider the number of function calls among modules across different RUs.

  The data dependencies are proportional to the size of the shared variables among modules across different RUs. In some of the previous work on local recovery ([16, 53]) it has been assumed that the RUs are stateless and as such do not contain shared state variables. This assumption can hold, for example, for stateless Internet service components [16] and stateless device drivers [53]. However, when an existing system is decomposed into RUs, there might be shared state variables leading to data dependencies between RUs. In fact, there have been also work focusing on saving component states in case of a component failure [68] and determining a global system state based on possibly interdependent component states [17]. Obviously, the size of data dependencies complicate the recovery and create performance overhead because the shared data need to be kept synchronized after recovery. This makes the amount of data dependency between RUs an important criteria for selecting RUs.

It appears that there exists an inherent trade-off between the availability and performance criteria. On the one hand increasing availability will require to increase the number of RUs[1]. On the other hand increasing the number of RUs will introduce additional performance overhead because the amount of function dependencies and data dependencies will increase. Therefore, selecting decomposition alternatives requires their evaluation with respect to these two criteria and making the desired trade-off.

5.4 The Overall Process and the Analysis Tool

In this section we define the approach that we use for selecting a decomposition alternative with respect to local recovery requirements. Figure 5.4 depicts the main activities of the overall process in a UML [100] activity diagram. Hereby, the filled circle and the filled circle with a border represent the starting point and the ending point, respectively. The rounded rectangles represent activities and arrows (i.e. flows) represent transitions between activities. The beginning of parallel activities are denoted with a black bar with one flow going into it and several leaving it. The end of parallel activities are denoted with a black bar with several flows entering it and one leaving it. This means that all the entering flows must reach the bar before the transition to the next activity is taken. Alternative activities are reached through different flows that are stemming from a single activity. The activities of the overall process are categorized into five groups as follows.

[1]This is based on the assumption that the fault tolerance mechanism remains simple and fault-free. By increasing the number of RUs, an error of a module will lead to a failure of a smaller part of the system (the corresponding RU), while the other RUs can remain available. However, depending on the fault assumptions and the system, increasing the number of RUs can increase complexity and this, in turn can increase the probability of introducing additional faults.

- *Architecture Definition*: The activities of this group are about the definition of the software architecture by specifying basic modules of the system. The given architecture will be utilized to define the feasible decomposition alternatives that show the isolated RUs consisting of the modules. To prepare the analysis for finding the decomposition alternative each module in the architecture will be annotated with several properties. These properties are utilized as an input for the analytical models and heuristics during the *Decomposition Selection* process.

- *Constraint Definition*: Using the initial architecture we can in principle define the design space including the decomposition alternatives. As we have seen in section 5.3.1 the design space can easily get very large and unmanageable. In addition, not all theoretically possible decompositions are practically possible because modules in the initial architecture cannot be freely allocated to RUs due to domain and stakeholder constraints. Therefore, before generating the complete design space first the constraints will be specified. The type of constraints that we consider can *i)* limit the number of RUs, *ii)* prevent the allocation of some modules to same or separate RUs, *iii)* limit the performance overhead that can be introduced by a decomposition alternative based on provided measurements from the system. The activities of the *Constraint Definition* group involve the specification of such domain and stakeholder constraints.

- *Design Space Generation*: After the domain constraints are specified the possible decomposition alternatives will be defined. Each decomposition alternative defines a possible partitioning of the initial architecture modules into RUs. The number of these alternatives is computed based on the Stirling numbers as explained in section 5.3.1.

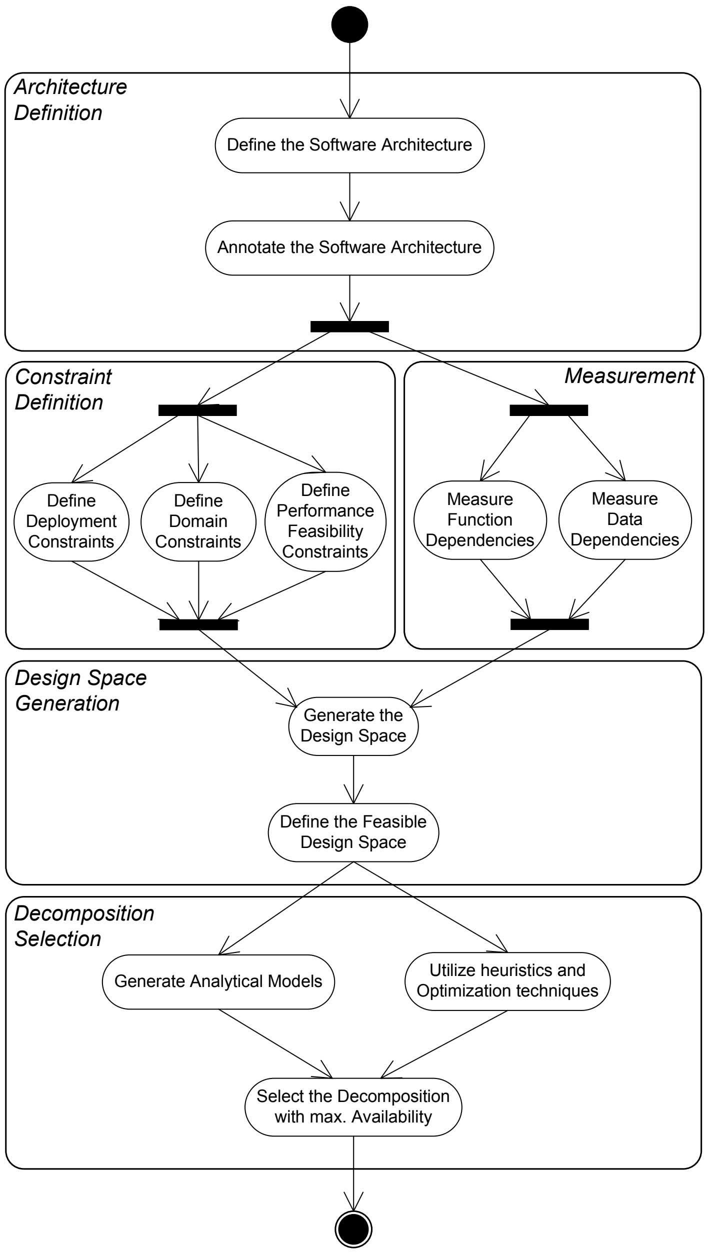

Architecture Definition
Define the Software Architecture
Annotate the Software Architecture
Constraint Definition
Define Deployment Constraints
Define Domain Constraints
Define Performance Feasibility Constraints
Measurement
Measure Function Dependencies
Measure Data Dependencies
Design Space Generation
Generate the Design Space
Define the Feasible Design Space
Decomposition Selection
Generate Analytical Models
Utilize heuristics and Optimization techniques
Select the Decomposition with max. Availability

Figure 5.4: The Overall Process

- *Performance Measurement*: Even though some alternatives are possible after considering the domain constraints they might in the end still not be feasible due to the performance overhead introduced by the particular allocation of modules to RUs. To estimate the real performance overhead of decomposition alternatives we need to know the amount of function and data dependencies among the system modules. The activities of the *Performance Measurement* group deal with performing related dynamic analysis and measurements on the existing system. These measurements are utilized during the *Decomposition Selection* process.

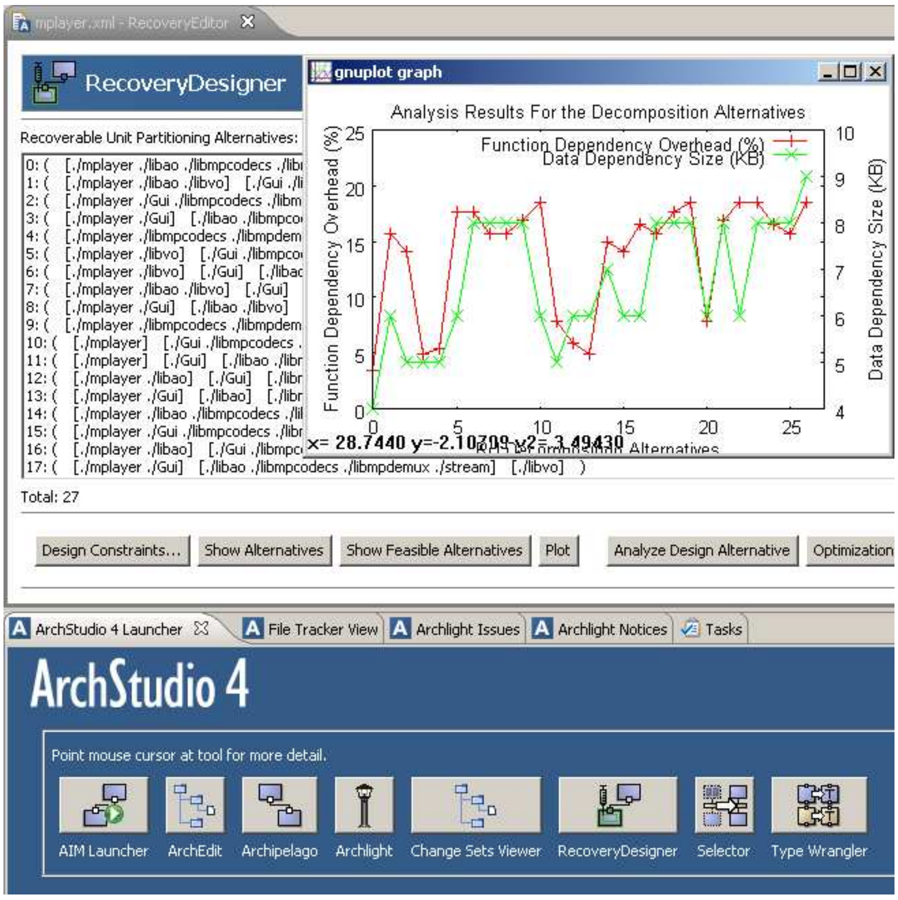
- *Decomposition Selection*: Based on the input of all the other groups of activities, the activities of this group select an alternative decomposition[2] based on either of two approaches. The selected approach depends on the size of the feasible design space derived from the architecture description, specified constraints and measurements in the previous activities. If the size is smaller than a maximum affordable amount, we generate analytical models for each alternative in the design space to estimate the system availability. Using the analytical models we select the RU decomposition alternative that is optimal with respect to availability and performance. If the design space is too large the generation of analytical models will be more time consuming. In that case we use a set of optimization algorithms and related heuristics to find a feasible decomposition alternative.

Figure 5.4 only shows a simple flow of the main activities in the overall process. There can be many iterations of these activities at different steps, starting from the architecture definition, constraint definition or decomposition selection. For brevity, we do not elaborate on this and we do not populate the diagram with these iterations.

We have developed an analysis tool, the *Recovery Designer*, that implements the provided approach. The tool is fully integrated in the *Arch-Studio* environment [23], which includes a set of tools for specifying and analyzing software architectures. Arch-Studio is an open-source software and systems architecture development environment based on the Eclipse open development platform [6].

We have used the meta-modeling facilities of ArchStudio to extend the tool and integrate the *Recovery Designer* with the provided default set of tools. A snapshot of the extended *Arch-Studio* can be seen in Figure 5.5 in which the *Recovery Designer* tool is activated. At the bottom of the tool a button *Recovery Designer* can be

[2]Although we use the term ‘decomposition’ for this activity, please note that the system is already decomposed into a set of modules. On top of this, we are introducing a decomposition by actually ‘composing’ the system modules into RUs.



Recoverable Unit Partitioning Alternatives:

```
0: (   [./mplayer ./libao ./libmpcodecs ./libı
1: (   [./mplayer ./libao ./libvo]   [./Gui ./li
2: (   [./mplayer ./Gui ./libmpcodecs ./libm
3: (   [./mplayer ./Gui]   [./libao ./libmpco
4: (   [./mplayer ./libmpcodecs ./libmpdem
5: (   [./mplayer ./libvo]   [./Gui ./libmpco
6: (   [./mplayer ./libvo]   [./Gui]   [./libac
7: (   [./mplayer ./libao ./libvo]   [./Gui]
8: (   [./mplayer ./Gui]   [./libao ./libvo]
9: (   [./mplayer ./libmpcodecs ./libmpdem
10: (   [./mplayer]   [./Gui ./libmpcodecs .
11: (   [./mplayer]   [./Gui]   [./libao ./libr
12: (   [./mplayer ./libao]   [./Gui]   [./libr
13: (   [./mplayer ./Gui]   [./libao]   [./libr
14: (   [./mplayer ./libao ./libmpcodecs ./lil
15: (   [./mplayer ./Gui ./libmpcodecs ./libr
16: (   [./mplayer ./libao]   [./Gui ./libmpc
17: (   [./mplayer ./Gui]   [./libao ./libmpcodecs ./libmpdemux ./stream]   [./libvo]   )
```

Total: 27

Figure 5.5: A Snapshot of the Arch-Studio Recovery Designer

seen, which opens the *Recovery Designer* tool consisting of a number of sub-tools. The architecture of *Recovery Designer* is shown in Figure 5.6. The tool boundary is defined by the large rectangle with dotted lines. The tool consist itself of six sub-tools *Constraint Evaluator*, *Design Alternative Generator*, *Function Dependency Analyzer*, *Data Dependency Analyzer*, *Analytical Model Generator*, and *Optimizer*. Further, it uses five external tools *Arch-Edit*, *GNU-gprof*, *Valgrind*, *gnuplot* and *CADP* model checker. In the subsequent sections, we will explain the activities of the proposed approach in detail and we will refer to the related component(s) of the tool.

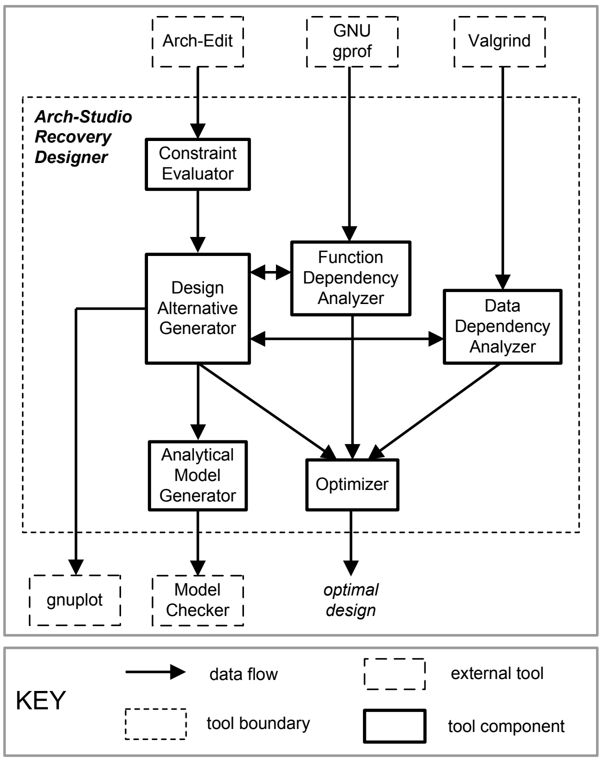


Figure 5.6: Arch-Studio Recovery Designer Analysis Tool Architecture

5.5 Software Architecture Definition

The first step *Architecture Definition* in Figure 5.4 is composed of two activities; *definition of the software architecture* and *annotation of the software architecture*. For describing the software architecture, we use the module view of the system, which includes basic implementation units and direct dependencies among them [18]. This activity is carried out by using the *Arch-Edit* tool of the *Arch-Studio* [23]. *Arch-Studio* specifies the software architecture with an XML-based architecture description language called *xADL* [23]. *Arch-Edit* is a front-end that provides a graphical user interface for creating and modifying an architecture description. Both the XML structure of the stored architectural description and the user interface of *Arch-Edit* (i.e. the set of properties that can be specified per module) conforms to the xADL schema and as such can be interchanged with other XML-based tools.

In the second activity of *Architecture Definition* a set of properties per module are defined to prepare the subsequent analysis. These properties are *MTTF*, *MTTR* and *Criticality*. *Criticality* property defines how critical the failure of a module is with respect to the overall functioning of the system.

To be able to specify these properties in our tool *Recovery Designer*, we have extended the existing xADL schema with a new type of interface, i.e. *reliability interface*, for modules. A part of the corresponding XML schema that introduces the new properties is shown in Figure 5.7. The concept of an interface with analytical parameters has been borrowed from [49], where Grassi et al. define an extension to the xADL language to be able to specify parameters for performance analysis. They introduce a new type of interface that comprises two type of parameters; *constructive parameter* and *analytic parameter*. The analytic parameters are used for analysis purposes only. In our case, we have specified the reliability interface that consists of the three parameters *MTTF*, *MTTR* and *Criticality*[3]. Using *Arch-Edit* both the architecture and the properties per module can be defined. The modules together with their parameters are provided as an input to analytical models and heuristics that are used in the *Decomposition Selection* process.

```
<xsd:complexType name="ReliabilityInterface">
<xsd:complexContent>
 <xsd:extension base="Interface">
  <xsd:sequence>
   <xsd:element name="MTTF"
                type="reliability:mttf"
                minOccurs="1" maxOccurs="1"/>
   <xsd:element name="MTTR"
                type="reliability:mttr"
                minOccurs="1" maxOccurs="1"/>
   <xsd:element name="Criticality"
                type="reliability:criticality"
                minOccurs="1" maxOccurs="1"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

Figure 5.7: xADL schema extension for specifying analytical parameters related to reliability

Note that the values for properties *MTTF*, *MTTR* and *Criticality* need to be defined by the software architect. In principle there are three strategies that can be used to determine these property values:

- *Using Fixed values*: It can be assumed that all modules have the same properties. Accordingly, all values can be fixed to a certain value just to investigate the analysis results.
- *What-if analysis*: A range of values can be considered, where the values are varied and their effect is observed.

[3]Notation-wise, we have placed all the properties under the reliability interface. However, only MTTF is actually related to reliability. In principle, a separate analytical interface could be defined for each property.

- *Measurement or estimation of actual values*: Values can be specified based on actual measurements from the existing system or estimation based on historical data.

The selection of these strategies is dependent on the available knowledge about the domain and the analyzed system. The measurement or estimation of actual values is usually the most accurate way to define the properties. However, this might not be possible due to lack of access to the existing system and historical data. In that case, either fixed values and/or a what-if analysis can be used. The specification of values for some module properties can also be supported by special analysis methods, tools and techniques. For example, *Criticality* values for modules can be specified based on the outcome of an analysis method like SARAH (Chapter 3).

In our MPlayer case study, we have used fixed *MTTF* and *Criticality* values but specified *MTTR* values based on measurements from the existing system. For all the modules, the values for *MTTF* and *Criticality* are defined as $0.5hrs$ and 1, respectively. We have measured the *MTTR* values from the actual implementation by calculating the mean time it takes to restart and initialize each module in the MPlayer over 100 runs. Table 5.1 shows the measured *MTTR* values.

Table 5.1: Annotated *MTTR* values for the MPlayer case based on measurements (*MTTF*=$0.5hrs$; *Criticality*=1).

Modules	*MTTR (ms)*
Libao	480
Libmpcodecs	500
Demuxer	540
Mplayer	800
Libvo	400
Stream	400
Gui	600

5.6 Constraint Definition

After defining the software architecture, we can start searching for the possible decomposition alternatives that define the structure of RUs. As stated before in section 5.3.1, not all theoretically possible alternatives are practically possible. During the *Constraint Definition* activity the constraints are defined and as such the design space is reduced. We distinguish among the following three types of constraints:

- *Deployment Constraints*: An RU is a unit of recovery in the system and includes a set of modules. In general, the number of RUs that can be defined will also be dependent on the context of the system that can limit the number of RUs in the system. For example, if we employ multiple operating system processes to isolate RUs from each other, the number of processes that can be created can be limited due to operating system constraints. It might be the case that the resources are insufficient for creating a separate process for each RU, when there are many modules and all modules are separated from each other.

- *Domain Constraints*: Some modules can be required to be in the same RU while other modules must be separated. In the latter case, for example, an untrusted 3^{rd} party module can be required to be separated from the rest of the system. Usually such constraints are specified with *mutex* and *require* relations [66, 21].

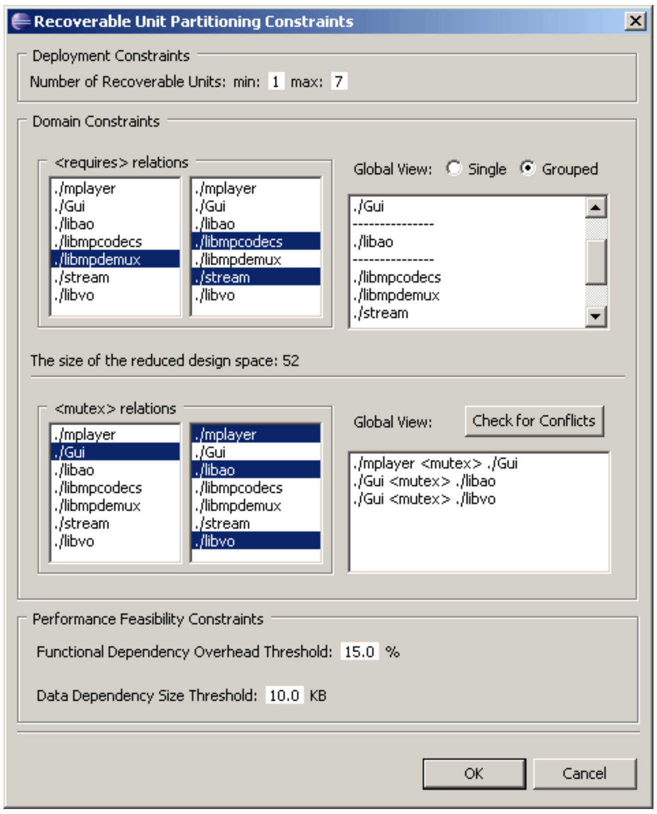
- *Performance Feasibility Constraints*: As explained in section 5.3.2, each decomposition alternative introduces a performance overhead due to function and data dependencies. Usually, there are thresholds for the acceptable amounts of these dependencies based on the available resources. The decomposition alternatives that exceed these thresholds are infeasible because of the performance overhead they introduce and as such must be eliminated.

The *Arch-Studio Recovery Designer* includes a tool called the *Constraint Evaluator*, where these constraints can be specified. *Constraint Evaluator* gets as input the architecture description that is created with the *Arch-Edit* tool (See Figure 5.6). We can see a snapshot of the *Constraint Evaluator* in Figure 5.8. The user interface consists of three parts: *Deployment Constraints*, *Domain Constraints* and *Performance Feasibility Constraints*, each corresponding to a type of constraint to be specified.

In the *Deployment Constraints* part, we specify the limits for the number of RUs. In Figure 5.8, the *minimum* and *maximum* number of RUs are specified as 1 and 7 (i.e. the total number of modules), respectively, which means that there is no limitation to the number of RUs.

In the *Domain Constraints* part we specify *requires/mutex* relations. For each of these relations, *Constraint Evaluator* provides two lists that include the name of the modules of the system. When a module is selected from the first list, the second list is updated, where the modules that are related are selected (initially, there are no selected elements). The second list can be modified by multiple (de)selection to change the relationships. For example, in Figure 5.8 it has been specified that

Demuxer must be in the same RU as *Stream* and *Libmpcodecs*, whereas *Gui* must be in a separate RU than *Mplayer*, *Libao* and *Libvo*. *Constraint Evaluator* can also automatically check if there are any conflicts between the specified *requires* and *mutex* relations (respectively two modules must be kept together and separated at the same time).



Recoverable Unit Partitioning Constraints

Deployment Constraints

Number of Recoverable Units: min: 1 max: 7

Domain Constraints

<requires> relations

./mplayer	./mplayer
./Gui	./Gui
./libao	./libao
./libmpcodecs	./libmpcodecs
./libmpdemux	./libmpdemux
./stream	./stream
./libvo	./libvo

Global View: Single [✓ Grouped]

```
./Gui
---------------
./libao
---------------
./libmpcodecs
./libmpdemux
./stream
```

The size of the reduced design space: 52

<mutex> relations

./mplayer	./mplayer
./Gui	./Gui
./libao	./libao
./libmpcodecs	./libmpcodecs
./libmpdemux	./libmpdemux
./stream	./stream
./libvo	./libvo

Global View: Check for Conflicts

```
./mplayer <mutex> ./Gui
./Gui <mutex> ./libao
./Gui <mutex> ./libvo
```

Performance Feasibility Constraints

Functional Dependency Overhead Threshold: 15.0 %

Data Dependency Size Threshold: 10.0 KB

Figure 5.8: Specification of Design Constraints

In the *Performance Feasibility Constraints* part, we specify thresholds for the amount of function and data dependencies between the separated modules. The specified constraints are used for eliminating alternatives that exceed the given threshold values. The dynamic analysis and measurements from the existing system that are performed to evaluate the *performance feasibility constraints* will be explained in section 5.8. However, if there is no existing system available and we can not perform the necessary measurements, we can skip the analysis of the *performance feasibility constraints*. *Arch-Studio Recovery Designer* provides an option to enable/disable the performance feasibility analysis. In case this analysis is disabled, the design space will be evaluated based on only the *deployment constraints* and *domain constraints* so that it is still possible to generate the decomposition alternatives and depict the reduced design space.

5.7 Design Alternative Generation

The first activity in *Generate Design Alternatives* is to compute the size of the feasible design space based on the architecture description and the specified domain constraints. *Design Alternative Generator* first groups the set of modules that are required to be together based on the *requires* relations[4] defined among the domain constraints. In the rest of the activity, *Design Alternative Generator* treats each such group of modules as a single entity to be assigned to a RU as a whole. Then, based on the number of resulting modules and the deployment constraints (i.e. the number of RUs), it uses the *Stirling numbers of the second kind* (section 5.3.1) to calculate the size of the reduced design space. Note that this information is provided to the user already during the *Constraint Definition* process (See the GUI of the *Constraint Evaluator* tool in Figure 5.8).

For the evaluation of the other constraints, the *Design Alternative Generator* uses the *restricted growth (RG) strings* [106] to generate the design alternatives. A RG string s of length n specifies the partitioning of n elements, where $s[i]$ defines the partition that i^{th} element belongs to. For the MPlayer case, for instance, we can represent all possible decompositions with a RG string of length 7. The RG string 0000000 refers to the decomposition where all modules are placed in a single RU. Assume that the elements in the string correspond to the modules *Mplayer*, *Libao*, *Libmpcodecs*, *Demuxer*, *Stream*, *Libvo* and *Gui*, then the RG string 010002 defines the decomposition that is shown in Figure 5.3. A *recursive lexicographic algorithm* generates all valid RG strings for a given number of elements and partitions [105].

During the generation process of decomposition alternatives, the *Design Alternative Generator* eliminates the alternatives that violate the *mutex* relations defined among the domain constraints. These are the alternatives, where two modules that must be separated are placed in the same RU. For the MPlayer case there exists a total of 877 decomposition alternatives. The deployment and domain constraints that we have defined reduced the number of alternatives down to 20.

[4]Note that the *requires* relation specifies a pair of modules that must be kept together in a RU. It does not specify a pair of modules that require services from each other. Such a dependency is not necessarily a reason for keeping modules together in a RU.

5.8 Performance Overhead Analysis

5.8.1 Function Dependency Analysis

Function calls among modules across different RUs impose an additional performance overhead due to the redirection of calls by RUs. In function dependency analysis, for each decomposition alternative, we analyze the frequency of function calls between the modules in different RUs. The overhead of a decomposition is calculated based on the ratio of the delayed calls to the total number of calls in the system.

Function dependency analysis is performed with the *Function Dependency Analyzer* tool based on the inputs from the *Design Alternative Generator* and *GNU gprof* tools (See Figure 5.6). As shown in Figure 5.9, the *Function Dependency Analyzer* tool itself is composed of three main components; *i*) *Module Function Dependency Extractor (MFDE) ii*) *Function Dependency Database* and *iii*) *Function Dependency Query Generator.*

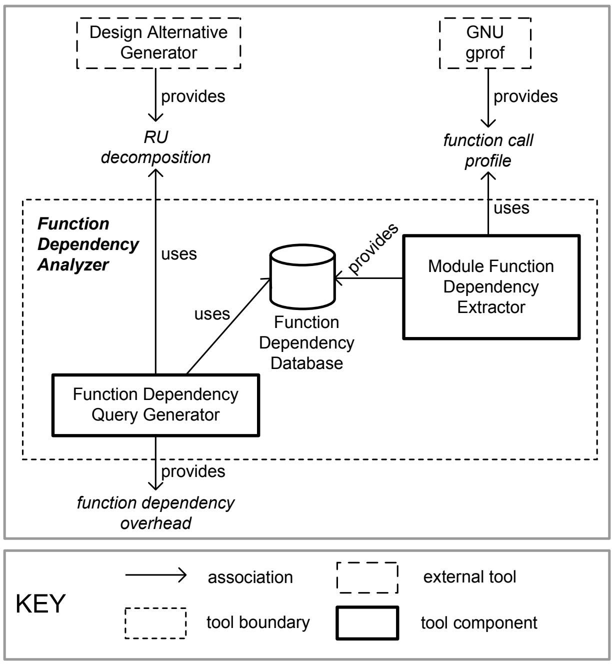

Design Alternative Generator
GNU gprof
provides
provides
RU decomposition
function call profile
uses
Function Dependency Analyzer
uses
provides
Module Function Dependency Extractor
uses
Function Dependency Database
Function Dependency Query Generator
provides
function dependency overhead
KEY
association
external tool
tool boundary
tool component

Figure 5.9: Function Dependency Analysis

MDFE uses the *GNU gprof* tool to obtain the function call graph of the system. *GNU gprof* also collects statistics about the frequency of performed function calls and the execution time of functions. Once the *function call profile* is available, MFDE uses an additional GNU tool called *GNU nm* (provides the symbol table of a C object file) to relate the function names to the C object files. As a result, MFDE creates a so-called Module Dependency Graph (MDG) [83]. MDG is a graph, where each node represents a C object file and edges represent the function dependencies between these files. Figure 5.10, for example, shows a partial snapshot of the generated MDG for MPlayer.

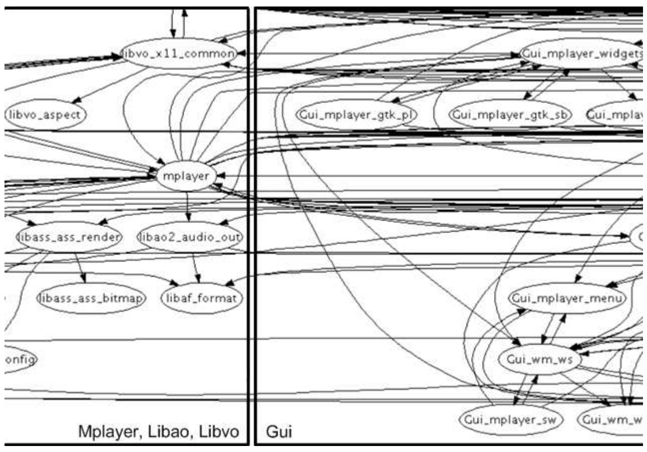


Figure 5.10: A Partial Snapshot of the Generated Module Dependency Graph of MPlayer together with the boundaries of the *Gui* module with the modules *Mplayer*, *Libao* and *Libvo*

To be able to query the function dependencies between the system modules, we need to relate the set of nodes in the MDG to the system modules. MFDE uses the folder (i.e. package) structure of the source code to identify the module that a C object file belongs to. This is also reflected to the prefixes of the nodes of the MDG. For example, the right hand side of Figure 5.10 shows some of the files that belong to the Gui module each of which has "*Gui*" as the prefix. MFDE exploits the full path of the C object file, which reveals its prefix (e.g. "./Gui*") and the corresponding module. For the MPlayer case, the set of packages that corresponds to the provided module view (Figure 5.3) were processed.

After the MDG is created, it is stored in the *Function Dependency Database*, which is a relational database. Once the MDG is created and stored, *Function Dependency Analyzer* becomes ready to calculate the function dependency overhead for a particular selection of RUs defined by the *Design Alternative Generator*. For each alternative, *Function Dependency Query Generator* accesses the *Function Dependency Database*, and automatically creates and executes queries to estimate the function dependency overhead. The function dependency overhead is calculated based on the following equation.

$$fd = \frac{\displaystyle\sum_{RUx}\sum_{\substack{RUy\wedge \\ (x\neq y)}} calls(x \rightarrow y) \times t_{OVD}}{\displaystyle\sum_{f} calls(f) \times time(f)} \times 100 \qquad (5.4)$$

In Equation 5.4, the denominator sums for all functions the number of times a function is called times the time spent for that function. In the nominator, the number of times a function is called among different RUs are summed. The end result is multiplied by the overhead that is introduced per such function call (t_{OVD}). In our case study, where we isolate RUs with separate processes, the overhead is related to the inter-process communication. For this reason, we use a measured worst case overhead, 100 ms as t_{OVD}.

Algorithm 1 Calculate the amount of function dependencies between the selected RUs

```
1: $sum \leftarrow 0$
2: for all RU $x$ do
3:    for all Module $m \in x$ do
4:       for all RU $y \wedge x \neq y$ do
5:          for all Module $k \in y$ do
6:             $sum \leftarrow sum + noOfCalls(m, k)$
7:          end for
8:       end for
9:    end for
10: end for
```

To calculate the function dependency overhead, we need to calculate the sum of function dependencies among all the modules that are part of different RUs. Consider, for example, the RU decomposition that is shown in Figure 5.3. Hereby, the *Gui* and *Libao* modules are encapsulated in separate RUs and separated from all the other modules of the system. The other modules in the system are allocated to a third RU, *RU MPCORE*. For calculating the function dependency overhead of this example decomposition, we need to calculate the sum of function dependencies between the *Gui* module and all the other modules plus the function dependencies between the *Libao* module and all the other modules. Algorithm 1 shows the pseudo code for counting the number of function calls betwen different RUs.

The procedure $noOfCalls(m, k)$ that is used in the algorithm creates and executes an SQL [123] query to get the number of calls between the specified modules m and

k. An example query is shown in Figure 5.11, where the number of calls from the module *Gui* to the module *Libao* is queried.

```
SELECT Sum(module_dependency.calls)
AS total FROM module_dependency
WHERE (module_dependency.src Like './Gui%'
And module_dependency.dest Like './libao%')
```

Figure 5.11: The generated SQL query for calculating the number of calls from the module *Gui* to the module *Libao*

For the example decomposition shown in Figure 5.3, the function dependency overhead was calculated as 5.4%. Note that this makes the decomposition a feasible alternative with respect to the function dependency overhead constraints, where the threshold was specified as 15% for the case study. It took the tool less than 100 ms. to calculate the function dependency overhead for this example decomposition. In general, the time it takes for the calculation depends on the number of modules and the particular decomposition. The worst case asymptotic complexity of Algorithm 1 is $O(n^2)$ with respect to the number of modules.

5.8.2 Data Dependency Analysis

Data dependency analysis is performed with the *Data Dependency Analyzer* (See Figure 5.6), which is composed of three main components; *i*) *Module Data Dependency Extractor (MDDE)* *ii*) *Data Dependency Database* and *iii*) *Data Dependency Query Generator* (See Figure 5.12).

MDDE uses the *Valgrind* tool to obtain the data access profile of the system. *Valgrind* [88] is a dynamic binary instrumentation framework, which enables the development of dynamic binary analysis tools. Such tools perform analysis at run-time at the level of machine code. *Valgrind* provides a core system that can instrument and run the code, plus an environment for writing tools that plug into the core system [88]. A *Valgrind tool* is basically composed of this core system plus the plug-in tool that is incorporated to the core system. We use *Valgrind* to obtain the data access profile of the system. To do this, we have written a plug-in tool for *Valgrind*, namely the *Data Access Instrumentor (DAI)*, which records the addresses and sizes of the memory locations that are accessed by the system. A sample of this data access profile output can be seen in Figure 5.13.

In Figure 5.13, we can see the addresses and sizes of memory locations that have been accessed. In line 7 for instance, we can see that there was a memory access at address *bea1d4a8* of size 4 from the file "audio_out.c". DAI outputs the full paths

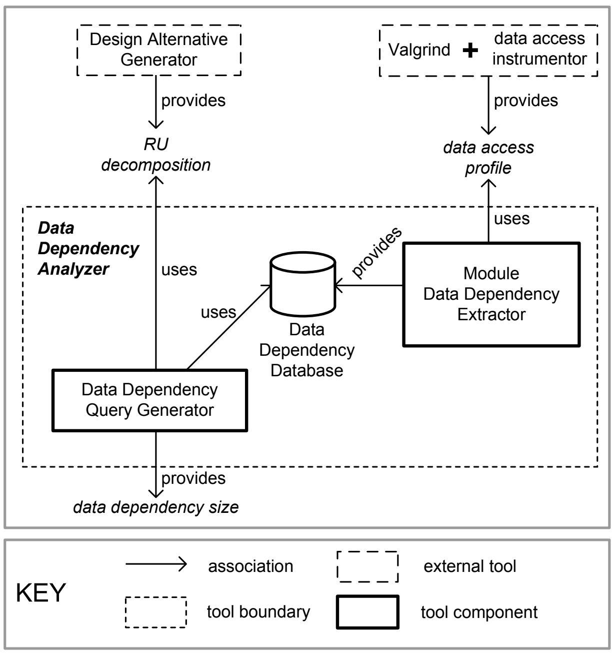


Figure 5.12: Data Dependency Analysis

of the files, which distinguishes to which module these files belong. The related parts of the file paths are underlined in Figure 5.13. For example, in line 9 we can see that the file “aclib_template.c” belongs to the *Libvo* module. From this data access profile we can also observe that the same memory locations can be accessed by different modules. Based on this, we can identify the data dependencies. For instance, Figure 5.13 shows a memory address *bea*97498 that is accessed by both the *Gui* and the *Mplayer* modules as highlighted in lines 2 and 6, respectively. The output of DAI is used by MDDE to search for all such memory addresses that are shared by multiple modules. The output of the MDDE is the total number and size of data dependencies between the modules of the system. Table 5.2 shows the

```
 1: ... /MPlayer-1.0rc1/Gui/interface.c: bea1d298,4;
 2: bea97498, 4; ... /MPlayer-1.0rc1/Gui/cfg.c:
 3: bea1d2a0, 4; 0862b714, 4; bea1d29c, 4; bea1d2a8,
 4: 4; bea1d2a4, 4; bea1d294, 4; bea1d288, 4; ...
 5: /MPlayer-1.0rc1/mplayer.c: bea1d4e8, 4; bea1d4e4,
 6: 4; bea1d4e0, 4; bea1d4dc, 4; bea97498, 4; ...
 7: /MPlayer-1.0rc1/libao2/audio_out.c: bea1d4a8, 4;
 8: bea1d4a4, 4; bea1d4a0, 4; bea1d49c, 4; bea1d498,
 9: 4; ... /MPlayer-1.0rc1/libvo/aclib_template.c:
10: 086b4860, 16; 086b4870, 16; ...
```

Figure 5.13: A Sample Output of Valgrind + Data Access Instrumentor

actual output for the MPlayer case.

Table 5.2: Measured data dependencies between the modules of the MPlayer

Module 1	*Module* 2	*count*	*size* (*bytes*)
Gui	*Libao*	64	256
Gui	*Libmpcodecs*	360	1329
Gui	*Libvo*	591	2217
Gui	*Demuxer*	47	188
Gui	*MPlayer*	36	144
Gui	*Stream*	242	914
Libao	*Libmpcodecs*	78	328
Libao	*Libvo*	63	268
Libao	*Demuxer*	3	12
Libao	*Mplayer*	43	172
Libao	*Stream*	54	232
Libmpcodecs	*Libvo*	332	1344
Libmpcodecs	*Demuxer*	29	116
Libmpcodecs	*Mplayer*	101	408
Libmpcodecs	*Stream*	201	812
Libvo	*Demuxer*	53	212
Libvo	*Mplayer*	76	304
Libvo	*Stream*	246	995
Demuxer	*Mplayer*	0	0
Demuxer	*Stream*	23	92
Mplayer	*Stream*	28	116

In Table 5.2 we see per pair of modules, the number of common memory locations accessed (i.e. count) and the total size of the shared memory. This table is stored in the *Data Dependency Database*. Once the data dependencies are stored, *Data Dependency Analyzer* becomes ready to calculate the data dependency size for a particular selection of RUs defined by the *Design Alternative Generator*. *Data Dependency Query Generator* accesses the *Data Dependency Database*, creates and executes queries for each RU decomposition alternative to estimate the data dependency size. The size of the data dependency is calculated simply by summing up the shared data size between the modules of the selected RUs. This calculation is depicted in the following equation.

$$dd = \sum_{RUx} \sum_{\substack{RUy \wedge \\ (x \neq y)}} \sum_{m \in x} \sum_{k \in y} memsize(m, k) \tag{5.5}$$

The querying process is very similar to the querying of function dependencies as explained in section 5.8.1. The only difference is that the information being queried is the data size instead of the number of function calls. For the example decomposition shown in Figure 5.3 the data dependency size is approximately 5 KB. It took the tool less than 100 ms. to calculate this. Note also that the decomposition turns out to be a feasible alternative with respect to the data dependency size constraints, where the threshold was specified as 10 KB for the case study.

5.8.3 Depicting Analysis Results

Using the deployment and domain constraints we have seen that for the MPlayer case the total number of 877 decomposition alternatives was reduced to 20. For each of these alternatives we can now follow the approach of the previous two subsections to calculate the function dependency overhead and data dependency size values. The *Arch-Studio Recovery Designer* generates a graph corresponding to the generated design space and uses *gnuplot* to present it. The tool allows to depict the generated design space at any time of the analysis process. In Figure 5.14 the total design space with the data dependency size and function dependency overhead values been plotted. The decomposition alternatives are listed along the *x-axis*. The *y-axis* on the left hand side is used for showing the function dependency overheads of these alternatives as calculated by the *Function Dependency Analyzer*. The *y-axis* on the right hand side is used for showing the data dependency sizes of the decomposition alternatives as calculated by the *Data Dependency Analyzer*. Figure 5.15 shows the plot for the 20 decomposition alternatives that remain after applying the deployment and domain constraints. Using these results the software architect can already have a first view on the feasible decomposition alternatives. The final selection of the alternatives will be explained in the next section.

5.9 Decomposition Alternative Selection

After the software architecture is described, design constraints are defined and the necessary measurements are performed on the system, the final set of decomposition alternatives can be selected as defined by the last group of activities (See Figure 5.4). Using the domain constraints we have seen that for the MPlayer case 20 alternatives were possible. This set of alternatives is further evaluated with respect to the performance feasibility constraints based on the defined thresholds and the measurements performed on the existing system. For the MPlayer case we have set the function dependency overhead threshold to 15% and the data dependency

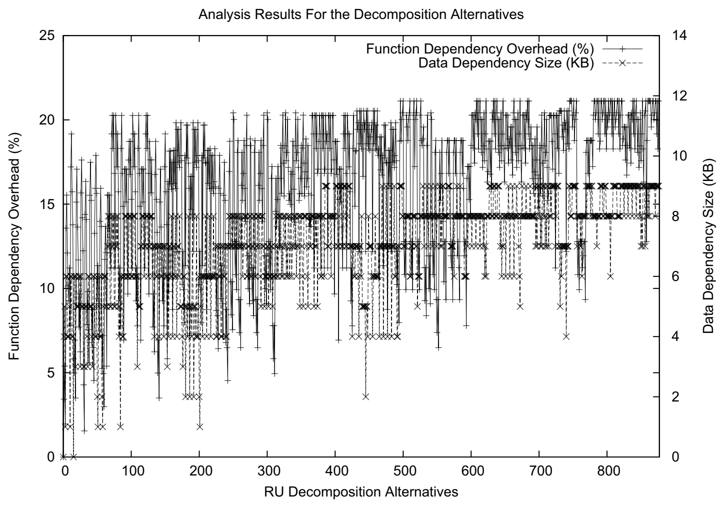


Figure 5.14: Function Dependency Overhead and Data Dependency Sizes for all Decomposition Alternatives

size threshold to 10 KB. It appears that after the application of performance feasibility constraints, only 6 alternatives are left in the feasible design space as listed in Figure 5.16. Hereby, RUs are defined in the straight brackets '[' and ']'. For example, the alternative # 1 represents the alternative as defined in Figure 5.3. Figure 5.17 shows the function dependency overhead and data dependency size for the 6 feasible decomposition alternatives. Hereby, we can see that the alternative # 4 has the highest value for the function dependency overhead. This is because, this alternative corresponds to the decomposition, where the two highly coupled modules *Libvo* and *Libmpcodecs* are separated from each other. We can see that this alternative has also a distinctively high data dependency size. This is because, this decomposition alternative separates the modules *Gui*, *Libvo* and *Libmpcodecs* from each other. As can be seen in Table 5.2, the size of data that is shared among these modules are the highest among all.

Since the main goal of local recovery is to maximize the system availability, we need to evaluate and compare the feasible decomposition alternatives based on availability as well. The approach adopted for the evaluation of availability and the selection of a decomposition depends on the size of the feasible design space. If the size is smaller than a maximum affordable amount depending on the available computation resources, we generate analytical models for each alternative in the design space. These analytical models are utilized for estimating the system availability. We select a RU decomposition based on the estimated availability and performance overhead of the feasible decomposition alternatives. If the design space is too large, we use a set of optimization algorithms to select one of the feasible decomposition

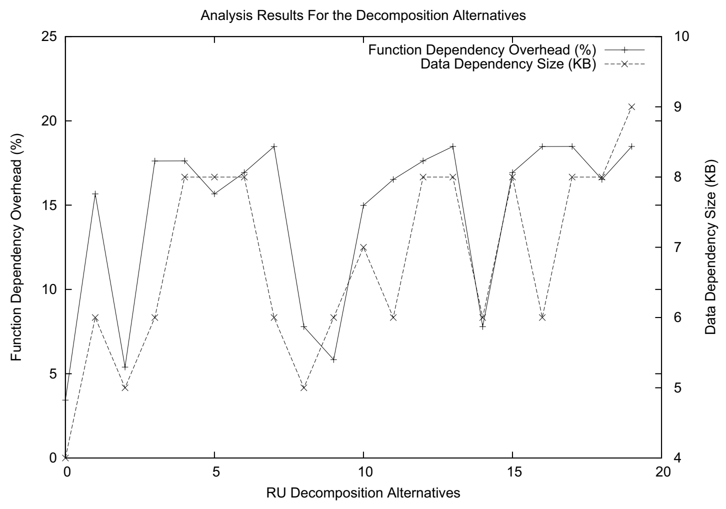


Figure 5.15: Function Dependency Overhead and Data Dependency Sizes for Decomposition Alternatives after domain constraints are applied

```
0: { [Mplayer, Libao, Libmpcodecs, Demuxer, Stream, Libvo] [Gui] }

1: { [Mplayer, Libmpcodecs, Demuxer, Stream, Libvo] [Gui] [Libao] }

2: { [Mplayer] [Gui] [Libao, Libmpcodecs, Demuxer, Stream, Libvo] }

3: { [Mplayer, Libao] [Gui] [Libmpcodecs, Demuxer, Stream, Libvo] }

4: { [Mplayer, Libao, Libmpcodecs, Demuxer, Stream] [Gui] [Libvo] }

5: { [Mplayer] [Gui] [Libao] [Libmpcodecs, Demuxer, Stream, Libvo] }
```

Figure 5.16: The feasible decomposition alternatives with respect to the specified constraints

alternative based on heuristics. These two approaches are explained in the following subsections.

5.10 Availability Analysis

In Figure 5.17, we can see the function dependency overhead and data dependency size for the 6 feasible decomposition alternatives. To select an optimal decomposition, we need to quantitatively assess these alternatives with respect to availability as well.

We exploit the compositional semantics of the I/O-IMC formalism to build availability models for alternative decompositions for local recovery. The *Analytical Model*

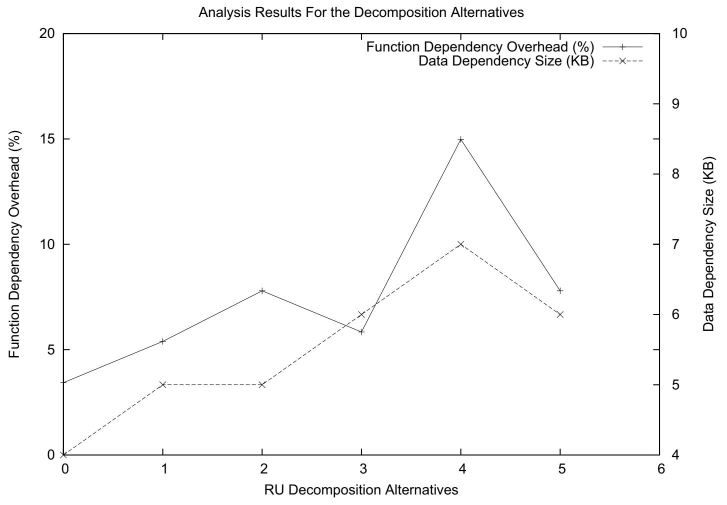


Figure 5.17: Function Dependency Overhead and Data Dependency Sizes for 6 Decomposition Alternatives

Generator automatically generates I/O-IMC models for the behavior of each module and RU of the system, and the coordination of recovery actions (See Appendix B). It also generates a script for the composition and analysis of these models. The execution of this script merges all the generated models in several iterations, pruning the state space in each iteration. The result of this composition and aggregation is a regular CTMC, which is analyzed to compute system availability. In the following, we explain our modeling approach and the analysis results.

5.10.1 Modeling Approach

In addition to the decomposition of system modules into RUs, the realization of local recovery (Chapter 6) requires *i*) control of communication between RUs and *ii*) coordination of recovery actions. These features can be implemented in several ways. In our modeling approach, we have introduced a model element, namely the *Recovery Manager (RM)*, to represent the coordination of recovery actions. The communication control between RUs is considered to be a part of the RM and as such is not modeled separately. In addition to the RM, we have introduced model elements for representing the behavior of individual modules and RUs. We have made the following assumptions in our modeling approach:

1. The failure of any module within an RU causes the failure of the entire RU.

2. The recovery of an RU entails the recovery (i.e. restart) of all of its modules

(even the ones that did not fail).

3. The failure of a module is governed by an exponential distribution (i.e. constant failure rate).
4. The recovery of a module is governed by an exponential distribution[5] and the error detection phase is ignored.
5. The RM does not fail.
6. As implied by the previous assumption, the RM always correctly detects a failing RU.
7. Only one RU can be recovered at a time and RM recovers the RUs on a *first-come-first-served (FCFS)* basis.
8. The recovery always succeeds.
9. The recovery of the modules inside a given RU is performed sequentially, one module at a time (a particular recovery sequence has no significance).

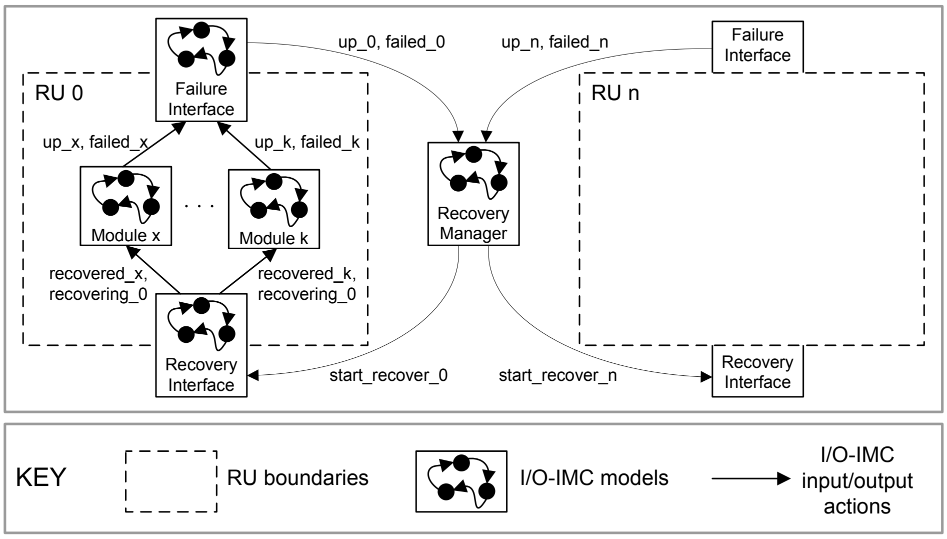


Figure 5.18: Modeling approach based on the I/O-IMC formalism

Figure 5.18 depicts our modeling approach. The RM only interfaces with RUs and is unaware of the modules within RUs. Each RU exhibits two interfaces; A

[5]This assumption was made for analytic tractability. An exponential distribution might not be, in some cases, a realistic choice; however, it is also possible to use a phase-type distribution, which approximates any distribution arbitrarily closely.

failure interface and a *recovery interface*. The failure interface essentially listens to the failure of the modules within the RU and outputs a *RU failure signal* upon the failure of a module. Correspondingly, the failure interface outputs an *RU up signal* upon the successful recovery of the all the modules. The RM listens to the *failure* and *up* signals emitted by the failure interfaces of the RUs. The recovery interface is responsible for the recovery of the modules within the RU. Upon the receipt of a *start_recover* signal from the RM, it starts a sequential recovery of the corresponding modules. Each module, recovery interface, failure interface, and the RM correspond to an I/O-IMC model as depicted in Figure 5.18. The exchanged signals are represented as input and output actions in these I/O-IMCs.

Analytical Model Generator first generates I/O-IMC models for each module. Based on the decomposition alternative provided by the *Design Alternative Generator*, it then generates the corresponding I/O-IMCs for failure/recovery interfaces and the RM. An example I/O-IMC model generated for the RM that controls two RUs is depicted in Figure 5.19. Appendix B contains detailed explanations of example I/O-IMCs (including the one depicted in Figure 5.19). It also gives details on the specification and generation process of I/O-IMC models.

In addition to generating all the necessary I/O-IMCs, a composition and analysis script is also generated, which conforms to the CADP SVL scripting language [42]. The execution of this script within CADP composes/aggregates all the I/O-IMCs based on the decomposition alternative, reduces the final I/O-IMC into a CTMC (See Appendix B for details), and computes the steady state probabilities of being at states, where the specified RUs are available. Based on these probabilities, the availability of the system is calculated as a whole. Figure 5.20 shows the generated CTMC for global recovery, where all modules are placed in a single RU.

In Figure 5.20, the initial state is *state* 0. It represents the system state, where the only RU of the system is available. All the other states represent the recovery stages of the RU. Since the RU includes 7 modules that are recovered (initialized) sequentially, there exist 7 states corresponding to the recovery of the RU. For example, *state* 7 represents the system state, where the RU is failed and non of its modules are recovered yet. There is a transition from *state* 0 to *state* 7, in which the label of the transition denotes the failure rate of this RU. Upon its failure, all the 7 modules comprised by the RU must be recovered. That is why, there exist 7 transitions from *state* 7 back to *state* 0, where the labels of these transitions specify the recovery rates of the modules. The steady state probability of being at *state* 0 was computed by CADP as 0.985880, which provides us the availability of the system as 98.5880%.

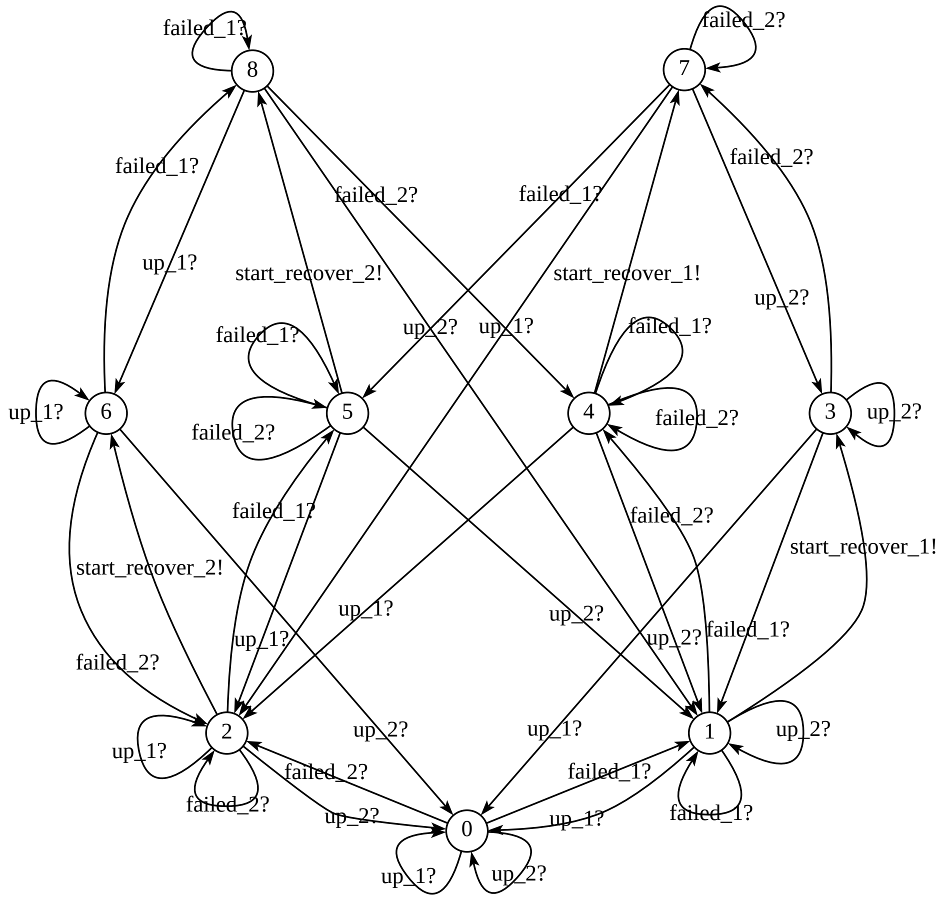


Figure 5.19: The generated I/O-IMC model for the Recovery Manager that controls two RUs

5.10.2 Analysis Results

We have generated CTMC models and calculated the availability for all the feasible decomposition alternatives that are listed in Figure 5.16. In this case study, we assume that the availability of the system is determined by the availability of the RU that comprises the *Mplayer* module. This is because, only the failure of this RU leads to the failure of the whole system regardless of the availability of the other RUs. The results are listed in Table 5.3 together with the corresponding function dependency overhead and data dependency size values as calculated before. The decomposition alternative # 1 represents the decomposition shown in Figure 5.3. The CTMC generated for this decomposition alternative comprises 60 states 122 transitions.

Based on the results listed in Table 5.3, we can evaluate and compare the feasible

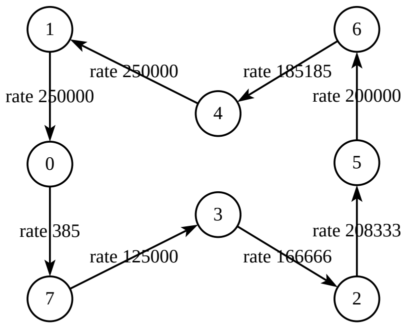


Figure 5.20: CTMC generated for global recovery, where all modules are placed in a single RU

Table 5.3: Estimated Availability, Function Dependency Overhead and Data Dependency Size for the Feasible Decomposition Alternatives

#	*Estimated Availability* (%)	*Function Dependency Overhead* (%)	*Data Dependency Size* (*bytes*)
0	98,9808	3,4350	4860
1	99,2791	5,3907	5860
2	99,9555	7,7915	5860
3	99,8589	5,8358	6516
4	99,2575	14,9803	7771
5	99,9557	7,7915	6688

decomposition alternatives. The decomposition alternatives # 0 and # 1 have the lowest function dependency overhead and data dependency size. That means that they introduce relatively less performance overhead and it is easier to implement them. However, the alternative # 0 has also the lowest availability among all the alternatives. We can also notice from the results that the decomposition alternative # 4 does not lead to a significantly higher availability as opposed to the high overhead it introduces. As will be explained in section 5.12, we have introduced local recovery to the MPlayer software for the decomposition alternatives # 0 and # 1.

5.11 Optimization

In our case study, we have evaluated a total of 6 feasible decomposition alternatives. Depending on the specified constraints and the number of modules, there can be too many feasible decomposition alternatives. In such cases it can take too much time to evaluate all the alternatives with analytical models although the models are generated automatically. Moreover, for some decomposition alternatives, the number of RUs can be too large. As a result, the state space of the model can grow too big despite of the reductions. Consequently, it might be infeasible for the model checker to evaluate the availability in an acceptable period of time.

For such cases, we also provide an option to select a decomposition among the feasible alternatives with heuristic rules and optimization techniques (See Figure 5.4). This activity is carried out with the *Optimizer* tool of *Arch-Studio Recovery Designer* as shown in Figure 5.6. The *Optimizer* tool employs optimization techniques to automatically select an alternative in the design space. The following objective function is adopted by the optimization algorithms based on heuristics.

$$MTTR_{RU_x} = \sum_{m \in RU_x} MTTR_m \tag{5.6}$$

$$1/MTTF_{RU_x} = \sum_{m \in RU_x} 1/MTTF_m \tag{5.7}$$

$$Criticality_{RU_x} = \sum_{m \in RU_x} Criticality_m \tag{5.8}$$

$$obj.func. = min. \sum_{RU\,RU_x} Criticality_{RU_x} \times \frac{MTTR_{RU_x}}{MTTF_{RU_x}} \tag{5.9}$$

In equations 5.6 and 5.7, we calculate for each RU the *MTTR* and *MTTF*, respectively. The calculation of *MTTR* for a RU is based on the fact that all the modules comprised by a RU are recovered sequentially. That is why the *MTTR* for a RU is simply the addition of *MTTR* values of the modules that are comprised by the corresponding RU. The calculation of *MTTF* for a RU is based on the assumption that the failure probability follows an exponential distribution with rate $\lambda = 1/MTTF$. If X_1, X_2, ... and X_n are independent exponentially distributed random variables with rates λ_1, λ_2, ... and λ_n, respectively, than $min(X_1, X_2, ..., X_n)$ is also exponentially distributed with rate $\lambda_1 + \lambda_2 + ... + \lambda_n$ [101]. As a result, the failure rate of a RU ($1/MTTF_{RU_x}$) is equal to the sum of failure rates of the modules that are comprised by the corresponding RU. In Equation 5.8, we calculate the *Criticality* of a RU by simply summing up the *Criticality* values of the modules that are comprised by the RU. Equation 5.9 shows the objective function that is utilized by the optimization algorithms. As explained in section 5.3.2, to maximize the availability, $MTTR$ of the system must be kept as low as possible and $MTTF$ of the system must be as high as possible. As a heuristic based on this fact, the objective function is to minimize the $MTTR/MTTF$ ratio in total for all RUs, which are weighted based on the *Criticality* values of RUs.

The *Optimizer* tool can use different optimization techniques to find the design alternative that satisfies the objective function the most. Currently it supports two optimization techniques; *exhaustive search* and *hill-climbing algorithm*. In this case of exhaustive search, all the alternatives are evaluated and compared with each other based on the objective function. If the design space is too large for exhaustive search, hill-climbing algorithm can be utilized to search the design space faster but ending up with possibly a sub-optimal result with respect to the objective function.

Hill-climbing algorithm starts with a random (potentially bad) solution to the problem. It sequentially makes small changes to the solution, each time improving it a little bit. At some point the algorithm arrives at a point where it cannot see any improvement anymore, at which point the algorithm terminates. The *Optimizer* utilizes the hill-climbing algorithm in a similar way to the clustering algorithm in [83]. The *Optimizer* starts from the worst decomposition with respect to availability, where all modules of the system are placed in a single RU. Then it systematically generates a set of *neighbor decomposition* by moving modules between RUs (called as a *partition* in [83]). A decomposition NP is defined as a neighbor decomposition of P if NP is exactly the same as P except that a single module of a RU in P is in a different RU in NP. During the generation process a new RU can be created by moving a module to a new RU. It is also possible to remove a RU when its only module is moved into another RU. The *Optimizer* generates neighbor decompositions of a decomposition by systematically manipulating the corresponding RG string. For example, consider the set of RG strings of length 7, where the elements in the string correspond to the modules *Mplayer*, *Libao*, *Libmpcodecs*, *Demuxer*, *Stream*, *Libvo* and *Gui*, respectively. Then the decomposition { [Libmpcodecs, Libvo] [Mplayer] [Gui] [Demuxer] [Libao] [Stream] } is represented by the RG string 1240350. By incrementing or decrementing one element in this string, we end up with the RG strings 1242350, 1243350, 1244350, 1245350, 1246350, 1240340, 1240351, 1240352, 1240353, 1240354, 1240355 and 1240356, which correspond to the decompositions shown in Figure 5.21, respectively.

For the MPlayer case, the optimal decomposition (ignoring the deployment and domain constraints) based on the heuristic-based objective function (Equation 5.9) is { [Mplayer] [Gui] [Libao] [Libmpcodecs, Libvo] [Demuxer] [Stream] }. It took 89 seconds for the *Optimizer* to find this decomposition with exhaustive search. The

```
 1: { [Libvo] [Mplayer, Libmpcodecs] [Gui] [Demuxer] [Libao] [Stream] }

 2: { [Libvo] [Mplayer] [Gui, Libmpcodecs] [Demuxer] [Libao] [Stream] }

 3: { [Libvo] [Mplayer] [Gui] [Libmpcodecs, Demuxer] [Libao] [Stream] }

 4: { [Libvo] [Mplayer] [Gui] [Demuxer] [Libao, Libmpcodecs] [Stream] }

 5: { [Libvo] [Mplayer] [Gui] [Demuxer] [Libao] [Libmpcodecs, Stream] }

 6: { [Libvo] [Mplayer] [Gui] [Demuxer] [Libao] [Stream] [Libmpcodecs] }

 7: { [Libmpcodecs, Libvo] [Mplayer] [Gui] [Demuxer] [Libao, Stream] }

 8: { [Libmpcodecs] [Mplayer, Libvo] [Gui] [Demuxer] [Libao] [Stream] }

 9: { [Libmpcodecs] [Mplayer] [Gui, Libvo] [Demuxer] [Libao] [Stream] }

10: { [Libmpcodecs] [Mplayer] [Gui] [Demuxer, Libvo] [Libao] [Stream] }

11: { [Libmpcodecs] [Mplayer] [Gui] [Demuxer] [Libao, Libvo] [Stream] }

12: { [Libmpcodecs] [Mplayer] [Gui] [Demuxer] [Libao] [Stream, Libvo] }

13: { [Libmpcodecs] [Mplayer] [Gui] [Demuxer] [Libao] [Stream] [Libvo] }
```

Figure 5.21: The neighbor decompositions of the decomposition { [Libmpcodecs, Libvo] [Mplayer] [Gui] [Demuxer] [Libao] [Stream] }

hill-climbing algorithm terminated on the same machine in 8 seconds with the same result. A total of 76 design alternatives had to be evaluated and compared by the hill-climbing algorithm, instead of 877 alternatives in the case of exhaustive search.

Selection of decomposition alternatives with heuristic rules and optimization techniques is a viable approach for evaluating large design spaces although the evaluation of alternatives might lead to different results than the evaluation based on analytical models. For example, under the consideration of the specified constraints in the MPlayer case study, the decomposition alternative # 5 as listed in Figure 5.16 has been selected as the optimal decomposition. This is indeed one of the best alternatives though the results in Table 5.3 show that there is a small difference of availability especially compared to the alternative # 2. In this case, we have rather selected the alternative #2 because of its relatively less function dependency overhead and smaller data dependency size.

5.12 Evaluation

The utilization of analytical models and optimization techniques helps us to analyze, compare and select decompositions among many alternatives. To assess the correctness and accuracy of these techniques, we have performed real-time measurements from systems that are decomposed for local recovery. In this section, we explain the performed measurements and the results of our assessments.

To implement local recovery, we have used a framework FLORA (Chapter 6). We have implemented local recovery for a total of 3 decomposition alternatives of MPlayer. *i*) Global recovery, where all the modules are placed in a single RU ({ [Mplayer, Libmpcodecs, Libvo, Demuxer, Stream, Gui, Libao] }) *ii*) Local recovery with two RUs, where the module *Gui* is isolated from the rest of the modules ({ [Mplayer, Libmpcodecs, Libvo, Demuxer, Stream, Libao] [Gui] }) *iii*) Local recovery with three RUs, where the module *Gui*, *Libao* and the rest of the modules are isolated from each other ({ [Mplayer, Libmpcodecs, Libvo, Demuxer, Stream, Libao] [Gui] }). Note that the 2^{nd} and the 3^{rd} implementations correspond to the decomposition alternatives # 0 and # 1 in Figure 5.16, respectively. We have selected these decomposition alternatives because they have the lowest function dependency overhead and data dependency size.

To be able to measure the availability achieved with these three implementations, we have modified each module so that they fail with a specified failure rate (assuming an exponential distribution with mean *MTTF*). After a module is initialized, it creates a thread that is periodically activated every second to inject errors. The operation of the thread is shown in Algorithm 2.

Algorithm 2 Periodically Activated Thread for Error Injection

```
 1: $time\_init \leftarrow currentTime()$
 2: while TRUE do
 3:    $time\_elapsed \leftarrow currentTime() - time\_init$
 4:    $p \leftarrow 1 - 1/e^{time\_elapsed/MTTF}$
 5:    $r \leftarrow random()$
 6:    if $p \geq r$ then
 7:       $injectError()$
 8:       $break$
 9:    end if
10: end while
```

The error injection thread first records the initialization time (Line 1). Then, each time it is activated, the thread calculates the time elapsed since the initialization (Line 3). The *MTTF* value of the corresponding module and the elapsed time is used for calculating the probability of error occurrence (Line 4). In Line 5, *random*() returns, from a uniform distribution, a sample value $r \in [0, 1]$. This value is compared to the calculated probability to decide whether or not to inject an error (Line 6). Possibly an error is injected by basically creating a fatal error with an illegal memory operation. This error crashes the process, on which the module is running (Line 7).

The *Recovery Manager* component of FLORA (Chapter 6) logs the initialization and failure times of RUs to a file during the execution of the system. For each of the implemented alternatives, we have let the system run for 5 hours. Then, we have processed the log files to calculate the times, when the core system module, *Mplayer* has been down. We have calculated the availability of the system based on the total time that the system has been running (5 hours). For the error injection, we have used the same $MTTF$ values ($0.5hrs$) as utilized by the analytical models and heuristics as explained in section 5.5. Table 5.4 lists the results for the three decomposition alternatives.

Table 5.4: Comparison of the estimated availability with analytical models, the heuristic outcome and the measured availability.

Decomposition Alternative	Measured Availability	Estimated Availability	Heuristic Objective Function
all modules in 1 RU	97.5732	98.5880	6.72
Gui, *the rest*	97.5834	$98,9808$	9.38
Gui, *Libao*, *the rest*	97.7537	$99,2791$	12.57

In Table 5.4, the first column shows the availability measured from the running systems. The second column shows the availability estimated with the analytical models for the corresponding decomposition alternatives. The third column shows the objective function value calculated to compare these alternatives during the application of optimization techniques. Here, we see that the measured availability and the estimated availability are quite close to each other. In addition, the design alternatives are ordered correctly with respect to their estimated availabilities, measured availabilities and the objective function used for optimization.

To amplify the difference between the decomposition alternatives with respect to availability, and such evaluate the results based on analytical models better, we have repeated our measurements and estimations for varying $MTTF$ values. As shown in Table 5.5, we have fixed $MTTF$ to $1800s$ ($= 0,5hrs$) for all the modules but *Libao* and *Gui*. For these two modules, we have specified $MTTF$ values as 60s and 30s, respectively.

Table 5.6 shows the results, when we have used the $MTTF$ values shown in Table 5.5

Table 5.5: $MTTF$ values that are used for the repeated estimations and measurements ($MTTR$ and *Criticality* values are kept as same as before).

Modules	*MTTF (s)*
Libao	60
Libmpcodecs	1800
Demuxer	1800
Mplayer	1800
Libvo	1800
Stream	1800
Gui	30

both for the analytical models and the error injection threads.

Table 5.6: Comparison of the estimated availability with analytical models, the heuristic outcome and the measured availability in the case of varying $MTTF$s.

Decomposition Alternative	Measured Availability	Estimated Availability	Heuristic Objective Function
all modules in 1 RU	83.27	83.60	0.5
Gui, *the rest*	92.31	93.25	1.24
Gui, *Libao*, *the rest*	97.75	98.70	2.83

Again, in Table 5.6, we observe that the design alternatives are ordered correctly with respect to their estimated availabilities, measured availabilities and the objective function used for optimization. We can also see that the measured availability and the estimated availability values (in %) are quite close to each other. In general, the measured availability is lower than the estimated availability. This is due in part to the communication delays in the actual implementation which are not accounted for in the analytical models. In fact, the various communications between the software modules, which are modeled using interactive transitions in the I/O-IMC models, abstract away any communicaion time delay (i.e. the communication is instantaneous). However, in reality, the recovery time includes the time for error detection, diagnosis and communication among multiple processes, which are subject to delays due to process context switching and inter-process communication overhead.

Based on these results, we can conclude that our approach can be used for accurately analyzing, comparing and selecting decomposition alternatives for local recovery.

5.13 Discussion

Partial failures

We have calculated the availability of the system according to the proportion of time that the core system module, *Mplayer* has been down. There are also partial failures, where *RU GUI* and *RU AUDIO* are restarted. In these cases, GUI panel vanishes and audio stops playing until the corresponding RUs are recovered. These are also failures from the user's perspective but we have neglected these failures to have a common basis for comparison with global recovery. Depending on the application, functional importance of the failed module and the recovery time, the user might get annoyed differently from partial failures. Also different users might have different perceptions. So, to take into account different type of partial failures, experiments are needed to be conducted to determine their actual effect on users [28]. In any case, the failure of the whole system would be the most annoying of all.

Frequency of data access

In data dependency analysis (Section 5.8.2), we have evaluated the size of shared data among modules. The shared data size is important since such data should be synchronized back after each recovery. The number of data access (column *count* in Table 5.2), on the other hand, is important due to redirection of the access through IPC. This is mostly covered by the function dependency analysis, where data is accessed through function calls. The correlation between function dependency overhead and data dependency size (Figure 5.15) shows this. In fact, our implementation of local recovery (explained in Chapter 6) requires that all data access must be performed through explicit function calls. For each direct data access without a function call, if there are any, the corresponding parts of the system are refactored to fullfill this requirement. After this refactoring, the function dependency analysis can be repeated to get more accurate results with respect to the expected performance overhead.

Specification of decomposition constraints

For defining the decomposition alternatives an important step is the specification of constraints. The more and the better we can specify the corresponding constraints the more we can reduce the design space of decomposition alternatives. In this work, we have specified deployment constraints (number of possible recoverable units), domain constraints and feasibility constraints (performance overhead thresholds).

The domain constraints are specified with *requires* and *mutex* relations. This has shown to be practical for designers who are not experts in defining complicated formal constraints. Nevertheless, a possible extension of the approach is to improve the expressiveness power of the constraint specification. For example, using first-order logic can be an option though in certain cases, the evaluation of the logical expressions and checking for conflicts can become NP-complete (e.g. the satisfiability problem). As such, workarounds and simplifications can be required to keep the approach feasible and practical.

Impact of usage scenarios on profiling results

The frequency of calls, execution times of functions and the data access profile can vary depending on the usage scenario and the inputs that are provided to the system. In our case study, we have performed our measurements for the video-playing scenario, which is a common usage scenario for a media player application. In principle, it is possible to take different types of usage scenarios into account. The results obtained from several system runs are statistically combined by the tools [40]. If there is a high variance among the execution of scenarios, where statistically combining the results would be wrong, multiple scenarios can be repeated in a period of time and the overhead can be calculated based on the profile information collected during this time period. However, this would require selection and prioritization of a representative set of scenarios with respect to their importance from the user point of view (user profile) [28]. The analysis process will remain the same although the input profile data can be different.

5.14 Related Work

For analyzing software architectures a broad number of architecture analysis approaches have been introduced in the last two decades ([19, 31, 46]). These approaches help to identify the risks, sensitivity points and trade-offs in the architecture. These are general-purpose approaches, which are not dedicated to evaluating the impact of software architecture decomposition on particular quality attributes. In this work, we have proposed a dedicated analysis approach that is required for optimizing the decomposition of architecture for local recovery in particular.

In [83] several optimization techniques are utilized to cluster the modules of a system. The goal of the clustering is to reverse engineer complex software systems by creating views of their structure. The part of our approach, where we utilize optimization techniques as explained in section 5.11 is very similar to and inspired from [83].

Although the underlying problem (set partitioning) is similar, we focus on different qualities. Our goal is to improve availability, whereas modularity is the main focus of [83]. As a result, we utilize different objective functions for the optimization algorithms.

There are several techniques for estimating reliability and availability based on formal models [121]. For instance, [72] presents a Markov model to compute the availability of a redundant distributed hardware/software system comprised of N hosts; [122] presents a 3-state semi-Markov model to analyze the impact of rejuvenation[6] on software availability and in [79], the authors use stochastic Petri nets to model and analyze fault-tolerant CORBA (FT-CORBA) [89] applications. In general however, these models are specified manually for specific system designs and the methodology lacks a comprehensive tool-support. Consequently, the process rather slow and error-prone, making these models less practical to use. In our tool, we generate formal models for different decomposition alternatives for local recovery automatically.

In this work, quantitative availability analysis is performed with Markov models, which are generated based on an architectural model expressed in an extended xADL. There are similar approaches that are based on different ADLs or, that use different analytical models. For example, architectural models that are expressed in AADL (Architecture Analysis and Design Language) has been used as a basis for generating Generalized Stochastic Petri Nets (GSPN) to estimate the availability of design alternatives [102]. UML based architectural models have also been used for generating analytical models to estimate dependability attributes. For instance, augmented UML diagrams are used in [80] for generating Timed Petri Nets (TPN).

A recoverable unit can be considered as a wrapper of a set of system modules to increase system dependability. Formal consistency analysis of error containment wrappers is performed in [65]. In this approach, system components and their composition are expressed based on the concepts of category theory. Using this specification, the propagation of data (wrong value) errors is analyzed and a globally consistent set of wrappers are generated based on assertion checks. The utilization of the iC2C (See section 4.8) is discussed in [25], where a method and a set of guidelines are presented for wrapping COTS (commercial off-the-shelf) components based on their undesired behavior to meet the dependability requirements of the system.

As far as local-recovery strategies are concerned, the work in [15] is similar to ours in the sense that several decomposition alternatives are evaluated for isolating system components from each other. However, their evaluation techniques are different: whereas we predict the availability at design time based on formal models, [15]

[6]Proactively restarting a software component to mitigate its aging and thus its failure.

uses heuristics to choose a decomposition alternative and evaluate it by running experiments on the actual implementation.

5.15 Conclusions

Local recovery requires the decomposition of software architecture into a set of isolated recoverable units. There exist many alternatives to select the set of recoverable units and each alternative has an impact on availability and performance of the system. In this chapter, we have proposed a systematic approach to evaluate such decomposition alternatives and balance them with respect to availability and performance. The approach is supported by an integrated set of tools. These tools in general can be utilized to support the analysis of a legacy system that is subject to major modification, porting or integration (e.g. with recovery mechanisms in this case). We have illustrated our approach by analyzing decomposition alternatives of MPlayer. We have seen that the performance overhead introduced by decomposition alternatives can be easily observed. We have utilized implementations of three decomposition alternatives to validate our analysis approach and the measurements performed on these systems have confirmed the validity of the analysis results.

The proposed approach requires the existence of the source code of the system for performance overhead estimation. This estimation is subject to the basic limitations of dynamic analysis approaches as described in section 5.1. The assumptions listed in section 5.10.1 constitute another limitation of the approach concerning the model-based availability analysis. The outcome of the analysis is also dependent on the provided *MTTF* values for the modules. The estimation of actual failure rates is usually the most accurate way to define the *MTTF* values. However, this requires historical data (e.g. a problem database), which can be missing or not accessible. In our case study, we have used fixed values and what-if analysis.

Chapter 6

Realization of Software Architecture Recovery Design

In the previous two chapters, we have explained how to document and analyze design alternatives for local recovery. After one of the design alternatives is selected, the software architecture should be structured accordingly. In addition, new supplementary architectural elements and relations should be implemented to enable local recovery. As a result, introducing local recovery to a system leads to additional development and maintenance effort. In this chapter, we introduce a framework called FLORA to reduce this effort. FLORA supports the decomposition and implementation of software architecture for local recovery.

The chapter is organized as follows. In the following section, we first outline the basic requirements for implementing local recovery. In seciton 6.2, we introduce the framework. Section 6.3 illustrates the application of FLORA with the MPlayer case study. We evaluate the application of FLORA in section 6.4. We conclude the chapter after discussing possibilities, limitations and related work.

6.1 Requirements for Local Recovery

Introducing local recovery to a system imposes certain requirements to its architecture. Based on the literature and our experiences in the TRADER [120] project we have identified the following three basic requirements:

- *Isolation*: An error (e.g. illegal memory access) occurring in one part of the system can easily lead to a system failure (e.g. crash). To prevent this failure and support local recovery we need to be able to decompose the system into a set of *recoverable unit (RU)* that can be isolated (e.g. isolation of the memory space). Isolation is usually supported by either the operating system (e.g. process isolation [59]) or a middleware (e.g. encapsulation of Enterprise Java Bean objects) and the existing design (e.g. *crash-only design*) [16].
- *Communication Control*: Although a RU is unavailable during its recovery, the other RUs might still need to access it in the mean time. Therefore, the communication between RUs must be captured to deal with the unavailability of RUs, for example, by queuing and retrying messages or by generating exceptions. Similar to loosely coupled components of the C2 architectural style [114], RUs should not directly communicate with each other. In [16], for instance, the communication is mediated by an application server. In general, various alternatives can be considered for realizing the communication control like completely distributed, hierarchical or centralized approaches.
- *System-Recovery Coordination*: In case recovery actions need to take place while the system is still operational, interference with the normal system functions can occur. To prevent this interference, the required recovery actions should be coordinated and the communication control should be leaded according to the applied recovery actions. Similar to communication control, coordination can also be realized in different ways ranging from completely distributed to completely centralized solutions.

6.2 FLORA: A Framework for Local Recovery

To reduce the development and maintenance effort for introducing local recovery while preserving the existing decomposition we have developed the framework FLORA. The framework has been implemented in the C language on a Linux platform (Ubuntu version 7.04) to provide a proof-of-concept for local recovery. FLORA assigns RUs to separate operating system (OS) processes, which do not directly communicate to each other. It relies on the capabilities of the platform to be able to

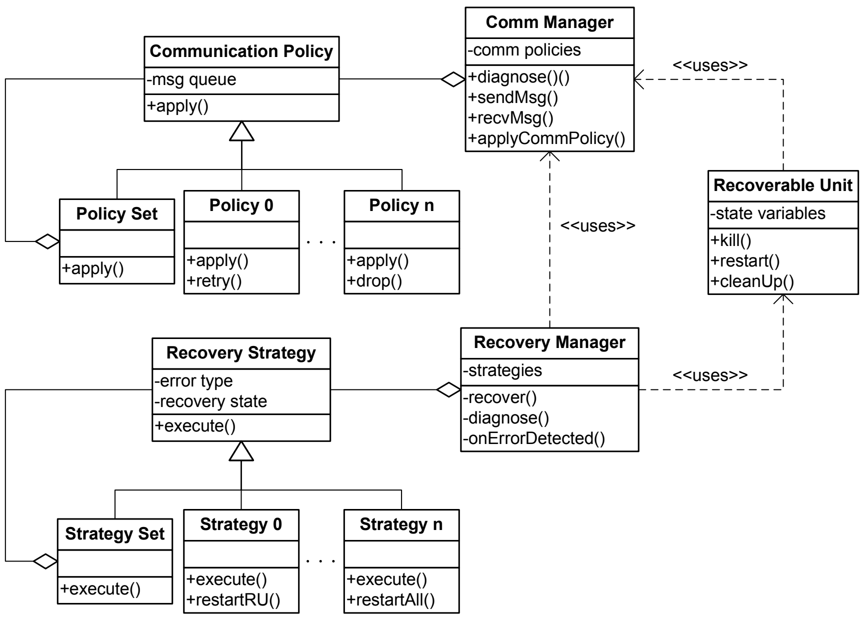


Figure 6.1: A Conceptual View of FLORA

control/monitor processes (kill, restart, detect a crash) and focuses only on transient faults. We assume that a process crashes or hangs in case of a failure of the RU that runs on it. FLORA restarts the corresponding RU in a new process and integrates it back to the rest of the system, while the other RUs can remain available.

The framework includes a set of reusable abstractions for introducing error detection, diagnosis, communication control between RUs. Figure 6.1 shows a conceptual view of FLORA as a UML class diagram. The framework comprises three main components: *Recoverable Unit (RU)*, *Communication Manager (CM)* and *Recovery Manager (RM)*.

To communicate with the other RUs, each RU uses the CM, which mediates all inter-RU communication and employs a set of communication policies (e.g. drop, queue, retry messages). Note that a communication policy can be composed of a combination of other primitive or composite policies. The RM uses the CM to apply these policies based on the executed recovery strategy. The RM also controls RUs (e.g. kill, restart) in accordance with the recovery strategy being executed. Recovery strategies can be composed of a combination of other primitive or composite strategies and they can have multiple states. They are also coupled with error types, from which they can recover. FLORA implements the detection of two type of errors: deadlock and fatal errors (e.g. illegal instruction, invalid data access). Each RU can detect deadlock errors via its wrapper, which detects if an expected response to a message is not received within a configured timeout period. The CM can discover the RU that fails to provide the expected message. Diagnosis information is conveyed to the RM, which kills and restarts the corresponding RU. RM can detect fatal errors. It is the parent process of all RUs and it receives a signal from the OS when a child process is dead. By handling this signal, the RM can also discover the failed RU based on the associated process identifier. Then it restarts the corresponding RU in a new process. If either of the CM or the RM fails, FLORA applies global recovery and restarts the whole system.

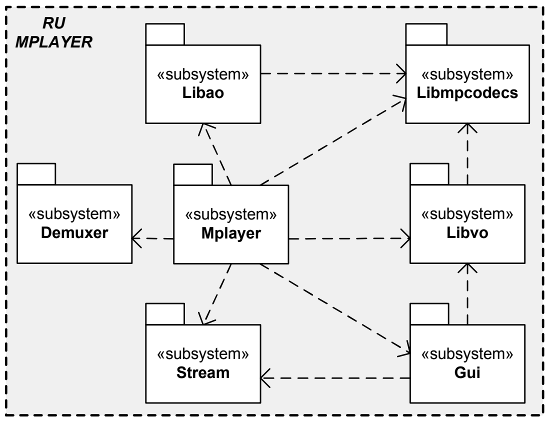
FLORA comprises Inter-Process Communication (IPC) utilities, message serialization / de-serialization primitives, error detection and diagnosis mechanisms, a RU wrapper template, one central RM and one central CM that communicate with one or more instances of RU. In the following section, we explain how FLORA can be applied to adapt a given architecture for local recovery.

6.3 Application of FLORA

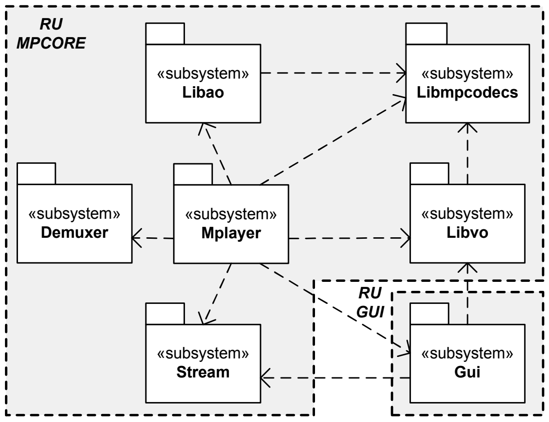
We have used FLORA to implement local recovery for 3 decomposition alternatives of MPlayer (See Figure 6.2). The implementations of these alternatives have been used in the evaluation of our analysis approach as presented in Chapter 5. The fist decomposition alternative places all the modules in a single RU (Figure 6.2(a)). In the second alternative, the module *Gui* and the rest of the modules are placed in different RUs (Figure 6.2(b)). The third decomposition alternative comprises 3 RUs, where the module *Gui*, *Libao* and the rest of the modules are separated from each other (Figure 6.2(c))

Figure 6.3 depicts the design of MPlayer for each of the decomposition alternatives after local recovery is introduced. Note that the figures 6.3(a) and 6.3(c) actually represent the same designs that were depicted before in the figures 4.7(a) and 4.7(b), respectively. In Chapter 4, the recovery style was used for depicting these design alternatives. In this chapter, we use the local recovery style to represent the same design alternatives in more detail.

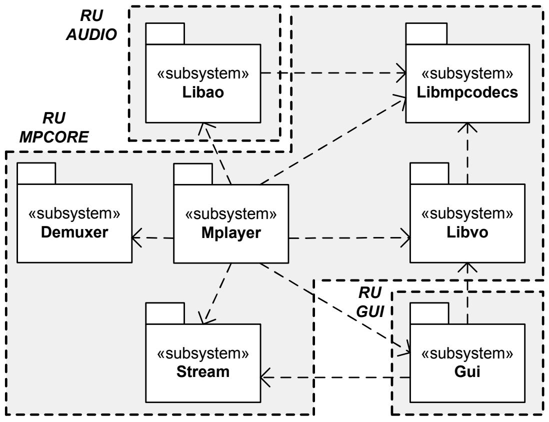
The first decomposition alternative shown in Figure 6.2(a) comprises only one RU. Hence, the CM is not used in this case and it does not take place in the corresponding recovery design shown in Figure 6.3(a). The third decomposition alternative was previously discussed in the previous two chapters and it was presented in the figures 4.6 and 5.3. In the corresponding recovery design, in Figure 6.3(c), we can see the 3 RUs, *RU MPCORE*, *RU GUI* and *RU AUDIO*. In addition, the components CM and RM have been introduced by the framework.



(a) all the modules are placed in one RU

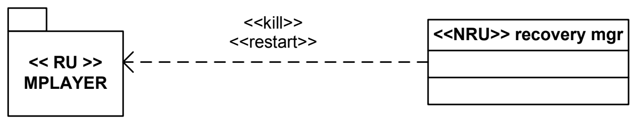


(b) *Gui* and the rest of the modules are placed in different RUs

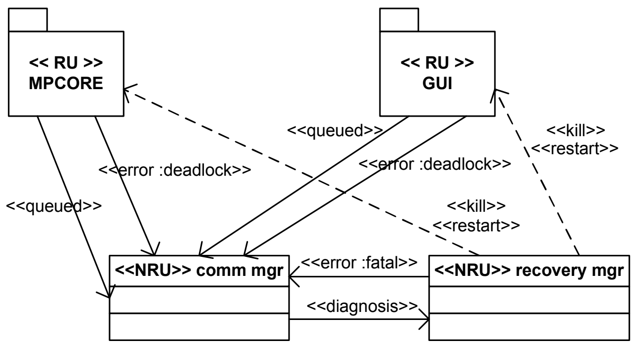


(c) *Gui*, *Libao* and the rest of the modules are placed in different RUs

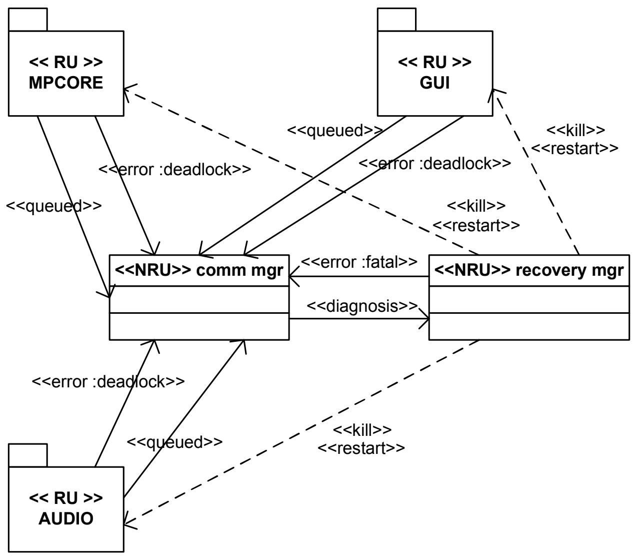
Figure 6.2: Realized decomposition alternatives of MPlayer (KEY: Figure 4.6)



(a) all the modules are placed in one RU



(b) *Gui* and the rest of the modules are placed in different RUs



(c) *Gui*, *Libao* and the rest of the modules are placed in different RUs

Figure 6.3: Recovery views of MPlayer based on the realized decomposition alternatives (KEY: Figure 4.4)

Each RU can detect deadlock errors. The RM can detect fatal errors. All error notifications are sent to the CM, which can control the communication accordingly. The CM conveys diagnosis information to the RM, which kills RUs and/or restarts dead RUs. Messages that are sent from RUs to the CM are stored (i.e. queued) by RUs in case the destination RU is not available and they are forwarded when the RU becomes operational again. On the other hand, RUs can also use stable storage utilities provided by FLORA to preserve some of their internal state after recovery. So, FLORA enables a combination of check-pointing and log-based recovery to preserve state.

To apply FLORA, each RU is wrapped using the RU wrapper template. Figure 6.4 shows a part of the RU wrapper that is used for *RU GUI*. It wraps and isolates the *Gui* module of MPlayer. The wrapper includes the necessary set of utilities for isolating and controlling a RU (lines 1-3). Hereby, *"rugui.h"* includes the list of function signatures, whereas *"util.h"* and *"recunit.h"* includes a set of standard utilities for RUs. Later, we will show and explain a part of the utilities provided by *"recunit.h"* that is related to check-pointing. To make use of this utility a set of state variables, their size, and format (integer, string, etc.) must be declared in the wrapper (lines 5-7). If needed, cleanup specific to the RU (e.g. allocated resources) can be specified (lines 10-12) as a preparation for recovery. Post-recovery initialization (lines 14-20) by default includes maintaining the connection with the CM and the RM (line 15), obtaining the check-pointed state variables (line 17) and start processing incoming messages from other RUs (line 19). Additional RU-specific initialization actions can also be specified here.

Each RU provides a set of interfaces, which are captured based on the specification in the wrapper (lines 22-26). Each interface defines a set of functions that are marshaled [32] and transferred through IPC. On reception of these calls, the corresponding functions are called and then the results are returned (lines 28-31 and 34-37). In all other RUs where this function is declared, function calls are redirected through IPC to the corresponding interface with C MACRO definitions. In principle we could also use *aspect-oriented programming techniques (AOP)* [37] for this, provided that an appropriate weaver for the C language is available. In Figure 6.5 a code section is shown from one of the modules of *RU MPCORE*, where all calls to the function *guiInit* are redirected to the function *mpcore_gui_guiInit* (line 1), which activates the corresponding interface (*INTERFACE_GUI*) instead of performing the function call (lines 4-6).

```
#include "util.h"
#include "recunit.h"
#include "rugui.h"
...
#define STATE_SIZE sizeof(int)
#define STATE_VARS guiIntfStruct.Playing
#define STATE_PARSE_FORMAT "d", &guiIntfStruct.Playing
...

void cleanUp() {
  /* no RU specific cleanup */
}

void __ruGui(struct recunit info) {
  INIT_RU(SOCK_PATH_RECMGR, SOCK_PATH_CONN)
  ...
  PRESERVE_STATE
  ...
  processMsgs();
}

void catchInterfaces() {
  BEGIN
    CATCH(INTERFACE_GUI, apOnMsgRcvd_gui)
  END
}

void OnMsgRcvd_gui_guiInit() {
  guiInit();
  RETURN(INTERFACE_GUI, msg_gui_guiInit)
}
...

void OnMsgRcvd_gui_guiGetFileName() {
  RETURN_ARGS(INTERFACE_GUI, msg_gui_guiGetFileName,
              sizeof(int) + (strlen(filename)+1),
              "ds", (strlen(filename)+1), filename)
  CHECKPOINT
}
...
```

Figure 6.4: RU Wrapper code for RU GUI

```
#define guiInit() mpcore_gui_guiInit()
...

void mpcore_gui_guiInit() {
  CALL(INTERFACE_GUI, msg_gui_guiInit)
}
...
```

Figure 6.5: Function redirection through RU interfaces

Figure 6.6 shows a part of the utilities provided by *"recunit.h"* as a set of C MACRO definitions. These utilities are imported by each RU wrapper by default (Line 2 in Figure 6.4). The definitions that are shown in Figure 6.6 are related to the check-pointing (Lines 3-4) and preservation of state (Lines 7-16). The *CHECKPOINT* MACRO sends the data that is defined in the RU wrapper (Line 6 in Figure 6.4) to the stable storage. Stable storage runs as a separate process in FLORA and it accepts get/set messages for retaining/saving data. The check-pointing locations are defined in the RU wrapper by the designer (Line 38 in Figure 6.4) based on the message exchanges through the RU interface. The *PRESERVE_STATE* MACRO obtains the saved data from the stable storage. The internal state of the RU is then updated based on the specified format (Line 7 in Figure 6.4), which defines a mapping of the saved data to the internal state variables.

```
 1: ...
 2:
 3: #define CHECKPOINT \
 4: CALL_ARGS(INTERFACE_SS_CONTROL, ..., STATE_VARS)
 5: ...
 6:
 7: #define PRESERVE_STATE \
 8: iRcvdStateSize = 0; \
 9: pcStateData = (char *)malloc(0); \
10: CALL(INTERFACE_SS_CONTROL, msg_ss_getData) \
11: parseMsgArgs(..., &iRcvdStateSize, &pcStateData); \
12: if(iRcvdStateSize > 0) \
13: { \
14:   parseMsgArgs(pcStateData, STATE_PARSE_FORMAT); \
15: } \
16: free(pcStateData);
17: ...
```

Figure 6.6: Standard RU utilities defined in *recunit.h* concerning check-pointing and state preservation

Besides the definition of RU wrappers, the designer should configure FLORA according to the recovery design. For example, Figure 6.7 shows a part of the configuration file that was used for the recovery design shown in Figure 6.3(c). Basically, the following information is included in the configuration file presented in Figure 6.7.

- the set of RUs (Lines 2-5)
- the set of interfaces (Lines 8-11)
- the identifiers and initialization functions of RUs (Lines 14-17)
- for each RU, the destination (the other RUs, RM, CM) of messages that will be sent through the defined interfaces (Lines 22-27)

```
...
/* recoverable units */
#define RUMPCORE 3
#define RUGUI 4
#define RUAUDIO 5
...

/* interfaces */
#define INTERFACE_ERROR 0
#define INTERFACE_COMM_CONTROL 1
#define INTERFACE_RECOVERY_CONTROL 2
...

/* initialization */
...
arecunit[RUGUI].iRUid = RUGUI; \
arecunit[RUGUI].pfMain = __ruGui; \
...

/* binding */
...
BEGINRU(RUGUI) \
  BIND(INTERFACE_RECOVERY_CONTROL, RECMGR) \
  BIND(INTERFACE_ERROR, CONNECTOR) \
  BIND(INTERFACE_MP, RUMPCORE) \
  ...
ENDRU \
...
```

Figure 6.7: Configuration of FLORA

6.4 Evaluation

If all the function calls that pass the boundaries of RUs are defined, FLORA guarantees the correct execution and recovery of these RUs. However, the specification of the RU boundaries with the RU wrapper template requires an additional effort. The main effort is spent due to the definition of the RU wrappers. For the decomposition shown in Figure 4.6, we have measured this effort based on the lines of code (LOC) written for RU wrappers and the actual size of the corresponding RUs[1]. Table 6.1 shows the LOC for each RU (LOC_{RU}), LOC of its wrapper ($LOC_{RUwrapper}$) and their ratio ($(LOC_{RUwrapper}/LOC_{RU}) \times 100$).

As we can see from Table 6.1, we had to write approximately 1K LOC to apply FLORA for the third decomposition alternative. The LOC written for wrappers is negligible compared to the corresponding system parts that are wrapped. The size

[1]We have excluded the source code for the various supported codecs, which are encapsulated mostly in *Libmpcodecs*.

Table 6.1: LOC for the selected RUs (as shown in Figure 4.6), LOC for the corresponding wrappers and their ratio

	LOC_{RU}	$LOC_{RUwrapper}$	ratio
RU MPCORE	214K	463	0,22%
RU GUI	20K	345	1,72%
RU AUDIO	8K	209	2,61%
TOTAL	242K	1017	0,42%

of the wrapper becomes even less significant for bigger system parts. In fact, the wrapper size is independent of the size and internal complexity of the system part that is wrapped. This is because the wrapper captures only the interaction of a RU with the rest of the system.

In FLORA, we have considered only transient faults, which lead to a crash or hanging of a OS process. In this scope, there is a standard implementation for the RM, the CM, and the RU wrapper utilities regardless of the number of and type of RUs. For this reason, LOC for these components remains the same when we increase the number of RUs[2]. Hence, mainly the specification of RU wrappers causes the extra LOC that should be written to apply FLORA. To be able to estimate the LOC to be written for wrappers, we have used the following equation.

$$LOC_{total} = 30 \times |RU| + 15 \times \sum_{r \in RU} calls(r \to \not r) \tag{6.1}$$

Equation 6.1 estimates the LOC needs to be written for wrappers based on the following assumptions. There should be a wrapper for each RU with some default settings (Figure 6.4). Therefore, the equation includes a fix amount of LOC (30) times the number of RUs ($|RU|$). In addition, all function calls between RUs must be defined in the corresponding wrappers. For each such function call we add a fix amount of LOC (15) taking into account the code for redirection of the function, capturing and processing its arguments and return values. To calculate Equation 6.1, we have used MFDE (Section 5.8.1) for calculating the number of function calls between the selected RU boundaries. We have generated all the set of possible partitions for varying number of RUs. We have calculated Equation 6.1 for each possible partition and we determined the minimum and maximum LOC estimations with respect to the number of RUs. The results can be seen in Figure 6.8.

In Figure 6.8, we can see the range of LOC estimations with respect to the number

[2]The performance overhead of these components at run-time, of course, increases.

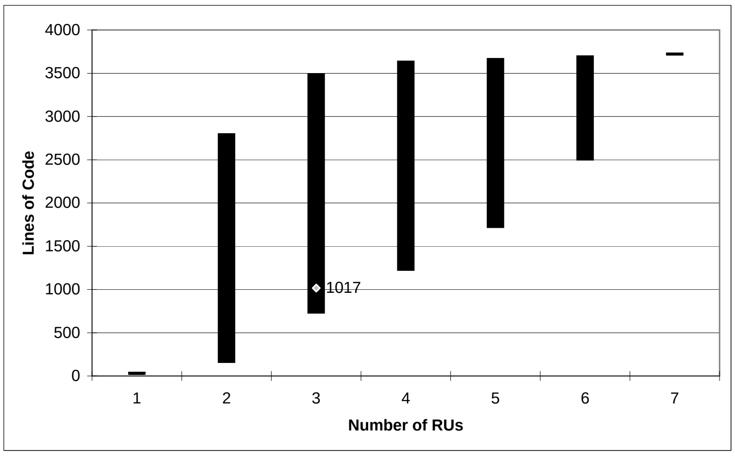


Figure 6.8: Estimated LOC to be written for wrappers with respect to the number of RUs

of RUs. When the number of RUs is equal to 1, there exists logically only one possible partitioning of the system. When the number of RUs is equal to 7, which is the number of modules in the system, there also exists only one possible partitioning of the system (each RU comprises a module of the system). Therefore, the minimum and the maximum values are equal for these cases. Figure 6.8 also marks the LOC written in the actual implemented case with 3 RUs as presented in Table 6.1. FLORA itself includes source code of size 1,5K LOC, which was reused as is. This includes the source code of the CM, the RM and standard utilities of RU wrappers (*"util.h"* and *"recunit.h"*). Thus, we can provide an approximate prediction of the effort for using the framework before the actual implementation.

Depending on how homogeneous the coupling between system modules is, the analysis can point out exceptional decompositions. For instance, in the analysis of the MPlayer case, we can see that several decompositions with 6 RUs require less overhead compared to the maximum overhead caused by a decomposition with 2 RUs. This means that there are certain modules that are exceptionally highly coupled (e.g. *Libvo* and *Libmpcodecs*) compared to the other modules.

6.5 Discussion

Refactoring the software architecture

FLORA aims at introducing local recovery to a system, while preserving the existing decomposition. To decide on the partitioning of modules into RUs, we can take coupling and shared data among the modules into account. The system can be analyzed to select a partitioning that requires minimal effort to introduce local recovery as discussed in Chapter 5 and in the previous section. On the other hand, one can also consider to refactor the software architecture to remove existing dependencies and shared data among some of the modules. The effort needed to refactor the architecture depends very much on the nature and semantics of the architecture. Restructuring for decoupling shared variables and removing function dependencies can be considered in case the architecture is already being refactored to improve certain qualities like reusability, maintainability or evolvability. In fact, this is a viable approach if the architects/developers have very good insight in the system. Otherwise, it would be better to treat modules as black boxes and wrap them with FLORA.

State preservation

State variables that are critical and need to be saved can be declared in RU wrappers. Declared state variables are check-pointed and automatically restored after recovery by FLORA. For instance, in the wrapper of *RU GUI* (See Figure 6.4), the state variable *guiInfStruct.Playing* is declared (Line 6). *guiInfStruct* is a C structure (*struct* [67]) and *Playing* is a variable within this structure, which keeps the status of the media player (e.g. playing, paused, stopped). The value of this variable is restored after the *Gui* module is restarted. However, particular states that are critical for modules are application dependent. Such states must be known and explicitly declared in the corresponding wrappers by the developers.

6.6 Related Work

In [26], a survey of approaches for application-level fault tolerance is presented. According to the categorization of this survey, FLORA falls into the category of *single-version software fault tolerance libraries (SV libraries)*. SV libraries are said to be limited in terms of separation of concerns, syntactical adequacy and adaptability [26]. On the other hand, they provide a good ratio of cost over improvement of the dependability, where the designer can reuse existing, long-tested and sophisticated pieces of software [26]. An example SV library is *libft* [58], which collects reusable software components for backward recovery. However, likewise other SV libraries [26] it does not support local recovery.

Candea et al. introduced the *microreboot* [16] approach, where local recovery is applied to increase the availability of Java-based Internet systems. Microreboot aims at recovering from errors by restarting a minimal subset of components of the system. Progressively larger subsets of components are restarted as long as the recovery is not successful. To employ microreboot, a system has to meet a set of architectural requirements (i.e. *crash-only design* [16]), where components are isolated from each other and their state information is kept in state repositories. Designs of many existing systems do not have these properties and it might be too costly to redesign and implement the whole system from the start. FLORA provides a set of reusable abstractions and mechanisms to support the refactoring of existing systems to introduce local recovery.

In [53], a micro-kernel architecture is introduced, where device drivers are executed on separate processes at user space to increase the failure resilience of an operating system. In case of a driver failure, the corresponding process can be restarted without affecting the kernel. The design of the operating system must support isolation between the core operating system and its extensions to enable such a recovery [53]. Mach kernel [96] also provides a micro-kernel architecture and flexible multiprocessing support, which can be exploited for failure resilience and isolation. Singularity [59] proposes multiprocessing support in particular to improve dependability and safety by introducing the concept of sealed process architecture, which limits the scopes of processes and their capabilities with respect to memory alteration for better isolation. To be able to exploit the multiprocessing support of the operating system for isolation, the application software must be partitioned to be run on multiple processes. FLORA supports the partitioning of an application software and reduces the re-engineering effort, while making use of the multiprocessing support of the Linux operating system.

Erlang/OTP (Open Telecoms Platform) [38] is a programming environment used by Ericsson to achieve highly available network switches by enabling local recovery. Erlang is a functional programming language and Erlang/OTP is used for (re)structuring Erlang programs in so-called *supervision trees*. A supervision tree assigns system modules in separate processes, called *workers*. These processes are arranged in a tree structure together with hierarchical *supervisor* processes at different levels of the tree. Each supervisor monitors the workers and the other supervisors among its children and restarts them when they crash. FLORA is used for restructuring C programs and partitioning them into several processes with a central supervisor (i.e. the RM).

We have implemented FLORA basically using macro-definitions in the C language. We could also implement FLORA using techniques [37] in which usually a distinction is made between *base code* on which additional so-called *crosscutting concerns* are *woven*. In particular the function redirection calls to IPC could be automatically woven using aspects. We consider this as a possible future work.

Some fault tolerance techniques for achieving high availability are provided by middlewares like FT CORBA [89]. However, it still requires additional development and maintenance efforts to utilize the provided techniques. Aurora Management Workbench (AMW) [13] uses model-centric development to integrate a software system with an high availability middleware. AMW generates code based on the desired fault tolerance behavior that is specified with a domain specific language. By this way, it aims at reducing the amount of hand-written code and as such reducing the developer effort to integrate fault tolerance mechanisms provided by the middleware with the system being developed. Developers should manually write code only for component-specific initialization, data maintenance and invocations of check-pointing APIs similar to those specified within RU wrapper templates. AMW allows software components (or servers) to be assigned to separate RUs (*capsule* in AMW terminology) and restarted independently. However, AMW currently does not support the restructuring and partitioning of legacy software to introduce local recovery.

6.7 Conclusions

In this chapter, we have presented the framework FLORA, which supports the realization of local recovery. We have used FLORA to implement local recovery for 3 decomposition alternatives of MPlayer. The realization effort for applying the framework appears to be relatively negligible. FLORA provides reusable abstractions for introducing local recovery to software architectures, while preserving their existing structure.

Chapter 7

Conclusion

Most home equipments today such as television sets are software-intensive embedded systems. The size and complexity of software in such systems are increasing with every new product generation due to the increasing number of new requirements, constantly changing hardware technologies and integration with other systems in networked environments. Together with time pressure, these trends make it harder, if not impossible, to prevent or remove all possible faults in a system. To cope with this challenge, fault tolerance techniques have been introduced, which enable systems to recover from failures of its components and continue operation. Many fault tolerance techniques are available but incorporating them in a system is not always trivial. In the following sections, we summarize the problems that we have addressed, the solutions that we have proposed and possible directions for future work.

7.1 Problems

In this thesis, we have addressed the following problems in designing a fault-tolerant system.

- *Early Reliability Analysis*: To select and apply appropriate fault tolerance techniques, we need reliability analysis methods to *i*) analyze potential failures, *ii*) prioritize them, and *iii*) identify sensitivity points of a system accordingly. In the consumer electronics domain, the prioritization of failures must be based on user perception instead of the traditional approach based on safety. Moreover, because implementing the software architecture is a costly process it is important to apply the reliability analysis as early as possible before committing considerable amount of resources.

- *Modeling Software Architecture for Recovery*: It turns out that introducing recovery mechanisms to a system has a direct impact on its architecture design, which results in several new elements, complex interactions and even a particular decomposition for error containment. Existing viewpoints [56, 69, 70] mostly capture functional aspects of a system and they are limited for explicitly representing elements, interactions and design choices related to recovery.

- *Optimization of Software Decomposition for Recovery*: Local recovery is an effective approach for recovering from errors. It enables the recovery of the erroneous parts of a system while the other parts of the system are operational. One of the requirements for introducing local recovery to a system is isolation. To prevent the failure of the whole system in case of a failure of one of its components, the software architecture must be decomposed into a set of isolated recoverable units. There exist many alternatives to decompose a software architecture for local recovery. Each alternative has an impact on availability and performance of the system. We need an adequate integrated set of analysis techniques to optimize the decomposition of software architecture for local recovery.

- *Realization of Local Recovery*: It appears that the optimal decomposition for local recovery is usually not aligned with the decomposition based on functional concerns and the realization of local recovery requires substantial development and maintenance effort. This is because additional elements should be introduced and the existing system must be refactored.

7.2 Solutions

In this section, we explain how we have addressed the aforementioned problems. The proposed techniques tackle different architecture design issues (modeling, analysis, realization) at different stages of software development life cycle (early analysis, maintenance) and as such are complementary.

7.2.1 Early Reliability Analysis

To select and apply appropriate fault tolerance techniques, in Chapter 3 we have introduced the software architecture reliability analysis method (SARAH). SARAH integrates the best practices of conventional reliability engineering techniques [34] with current scenario-based architectural analysis methods [31]. Conventional reliability engineering includes mature techniques for failure analysis and evaluation techniques. Software architecture analysis methods provide useful techniques for early analysis of the system at the architecture design phase. Despite most scenario-based analysis methods which usually do not focus on specific quality factors, SARAH is a specific purpose analysis method focusing on the reliability quality attribute. Further, unlike conventional reliability analysis techniques which tend to focus on safety requirements SARAH prioritizes and analyzes failures basically from the user perception because of the requirements of consumer electronics domain. To provide a reliability analysis based on user perception we have extended the notion of fault trees and refined the fault tree analysis approach. SARAH has been illustrated for identifying the sensitive modules for the DTV and it has provided an important input for the enhancement of the architecture. Besides the outcome of the analysis, the process of doing such an explicit analysis has provided better insight in the potential risks of the system.

7.2.2 Modeling Software Architecture for Recovery

We have introduced the recovery style in Chapter 4 to document and analyze recovery properties of a software architecture. We have also introduced a local recovery style and illustrated its application on the open source media player, MPlayer. Recovery views of MPlayer based on the recovery style have been used within our project to communicate the design of our local recovery framework (FLORA) and its application to MPlayer. These views have formed the basis for analysis and support for the detailed design of the recovery mechanisms.

7.2.3 Optimization of Software Decomposition for Recovery

We have proposed a systematic approach in Chapter 5 for analyzing an existing system to decompose its software architecture to introduce local recovery. The approach enables *i*) depicting the alternative space of the possible decomposition alternatives, *ii*) reducing the alternative space with respect to domain and stakeholder constraints, and *iii*) balancing the feasible alternatives with respect to availability and performance. We have provided the complete integrated tool-set, which supports the whole process of decomposition optimization. Each tool automates a particular step of the process and it can be utilized for other types of analysis and evaluation as well. We have illustrated our approach by analyzing decomposition alternatives of MPlayer. We have seen that the impact of decomposition alternatives can be easily observed, which are based on actual measurements regarding the isolated modules and their interaction. We have implemented local recovery for three decomposition alternatives to validate the results that we obtained based on analytical models and optimization techniques. The measured results turn out to be closely matching the estimated availability, while the alternatives were ordered correctly with respect to their estimated and measured availabilities and the objective function used for optimization.

7.2.4 Realization of Local Recovery

In Chapter 6 we have presented the framework FLORA that provides reusable abstractions to preserve the existing structure and support the realization of local recovery. We have used FLORA to implement local recovery for the three decomposition alternatives of MPlayer that are used for validating the analysis results based on analytic models and optimization techniques. In total three recoverable units (RUs) have been defined, which were overlaid on the existing structure without adapting the individual modules. In addition, the realization effort for applying the framework and introducing local recovery appears to be relatively negligible. The application of the framework, as such, provides a reusable and practical approach to introduce local recovery to software architectures.

7.3 Future Work

In the following, we provide several future directions regarding each of the methods and tools presented in this thesis.

The scenario derivation process in SARAH can be improved by utilizing a software architecture model with more semantics to be able to check properties of derived scenarios (e.g. whether they correctly capture the error propagation). The other inputs of the analysis can also be improved like severity values for failures based on user perception and fault occurrence probabilities. We have made use of spreadsheets to perform the required calculations in SARAH. A better tool-support, possibly with a visual editor, can be provided to reduce the analysis effort.

We have introduced the local recovery style as a specialization of the recovery style to represent a local recovery design in more detail. Similarly, we can introduce additional architectural styles to represent particular fault tolerance mechanisms. We have introduced the recovery style based on the module viewtype. As discussed in 4.7, we can also derive a recovery style for the component & connector viewtype or the allocation viewtype to represent recovery views of run-time or deployment elements.

Several points can be improved in the analysis and optimization of decomposition alternatives for recovery. As discussed in 5.13, we can increase the expressiveness power of the constraint specification. Another improvement can be about the estimation of function and data dependencies, for which we have performed our measurements during a single usage scenario. The analysis can take different types of scenarios into account based on the usage profile of the system. As for future directions regarding model-based availability analysis, one can revisit some or all of the assumptions made in section 5.10.1 and/or modify any of the four basic I/O-IMC models (Appendix B.1) to more accurately represent the real system behavior. In addition, new heuristics and alternative optimization algorithms can be incorporated in the tool-set for a faster or better (i.e. closer to the global optimum) selection of a decomposition alternative.

FLORA can be improved by reconsidering the fault assumptions and related fault tolerance mechanisms to be incorporated. FLORA is basically a software library and it still requires effort to utilize the library. This effort can be reduced by using model-driven engineering and aspect-oriented software development techniques as discussed in section 6.6.

As a future work concerning all the proposed techniques, further case studies or experiments can be conducted to evaluate the methods and tools within the context of various application domains.

Appendix A

SARAH Fault Tree Set Calculations

In the following, Figure A.1 depicts the Fault Tree Set that was used for the analysis presented in Chapter 3. Failure scenarios are labeled with the IDs of the associated architectural elements and the user perceived failures are annotated with severity values. Table A.1 shows the corresponding Architectural-Level analysis results. Hereby, AEID, NF, WNF, PF and WPF stand for Architectural Element ID, Number of Failures, Weighted Number of Failures, Percentage of Failures and Weighted Percentage of Failures, respectively.

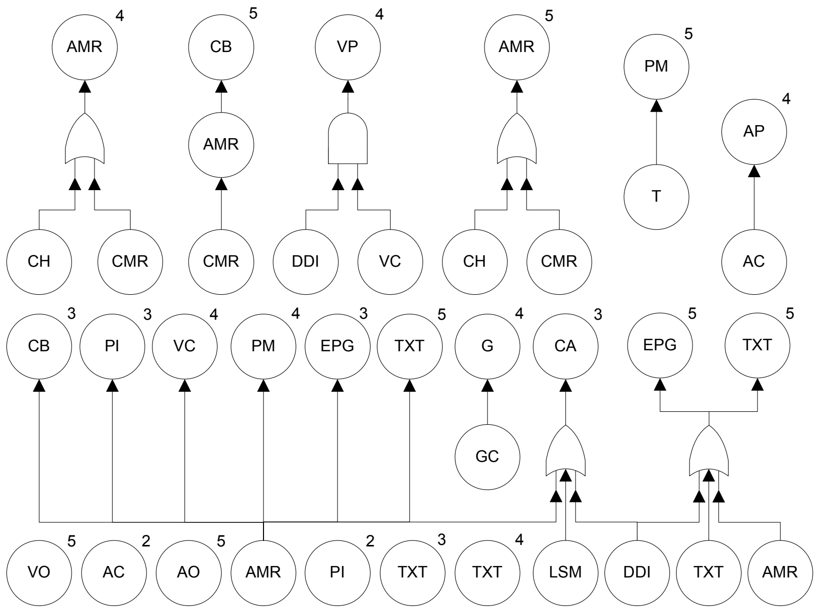


Figure A.1: Fault Tree Set

Table A.1: SARAH Architecture-Level Analysis results

AEID	AC	AMR	AO	AP	CA	CB	CH	CMR	DDI	EPG
NF	2	5	1	1	1	2	2	3	2	2
WNF	6	49	5	4	3	8	9	14	15	8
PF	5,41	13,51	2,70	2,70	2,70	5,41	5,41	8,11	5,41	5,41
WPF	3,09	25,26	2,58	2,06	1,55	4,12	4,64	7,22	7,73	4,12

AEID	G	GC	PI	PM	LSM	T	TXT	VO	VC	VP
NF	1	1	2	2	1	1	4	1	2	1
WNF	4	4	5	9	4	5	27	5	6	4
PF	2,70	2,70	5,41	5,41	2,70	2,70	10,81	2,70	5,41	2,70
WPF	2,06	2,06	2,58	4,64	2,06	2,58	13,92	2,58	3,09	2,06

Appendix B

I/O-IMC Model Specification and Generation

B.1 Example I/O-IMC Models

In this section, we provide details on the 4 basic I/O-IMC models that we have used for availability analysis of decomposition alternatives for local recovery. As briefly explained in section 5.10.1, these are the *module I/O-IMC*, the *failure interface I/O-IMC*, the *recovery interface I/O-IMC*, and the *recovery manager (RM) I/O-IMC*. To explain these models, we use the running example with 3 modules and 2 RUs as depicted in Figure B.1. This example consists of a RM and two recoverable units (RUs); *RU 1* has one module *A* and *RU 2* has two modules *B* and *C*. By convention, the starting state of any I/O-IMC is state 0 and the RUs are numbered starting from 1.

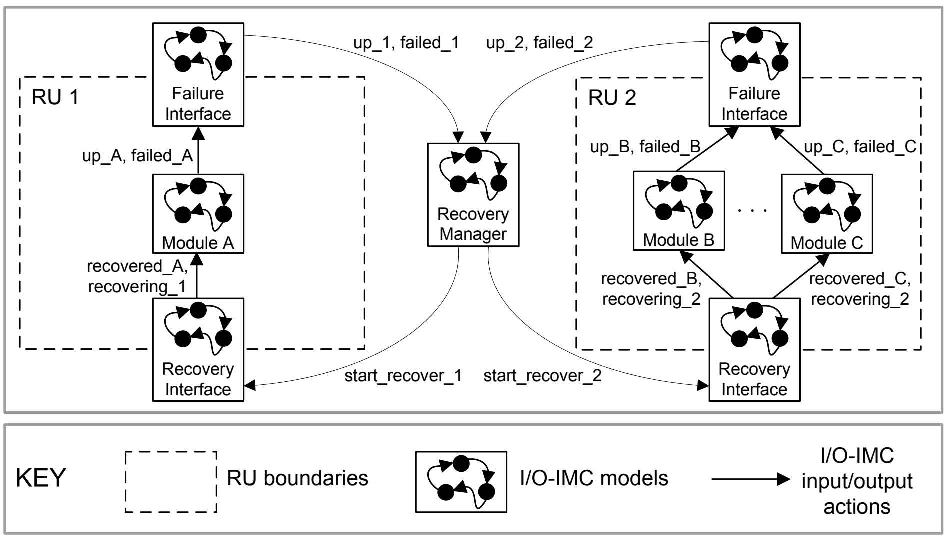


Figure B.1: The running example with 3 modules and 2 RUs

B.1.1 The module I/O-IMC

Figure B.2 shows the I/O-IMC of module *B*. The module is initially operational in state 0, and it can fail with rate 0.2 and move to state 2. In state 2, the module notifies the failure interface of *RU 2* about its failure (i.e. transition from state 2 to state 1). In state 1, the module awaits to be recovered (i.e. receiving signal 'recovered_B' from the recovery interface), and once this happens it outputs an 'up' signal notifying the failure interface about its recovery (i.e. transition from state 3 to state 0). Signal 'recovering_2' is received from the recovery interface indicating that a recovery procedure of *RU 2* has been initiated. The remaining input transitions are necessary to make the I/O-IMC *input-enabled*.

B.1.2 The failure interface I/O-IMC

Figure B.3 shows the I/O-IMC model of *RU 2* failure interface. The failure interface simply listens to the failure signals of modules *B* and *C*, and outputs a 'failure' signal for the RU (i.e. 'failed_2') upon the receipt of any of these two signals. In fact, this interface behaves as an OR gate in a fault tree. So, the failure interface outputs an 'up' signal for the RU (i.e. 'up_2') when all the failed modules have output their 'up' signals. For instance, consider the following sequence of states: 0, 1, 4, 7, and 0; this corresponds to modules *B* and *C* being initially operational, then *B* fails,

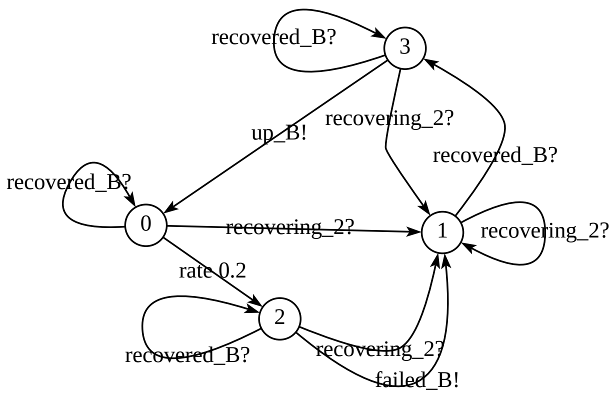


Figure B.2: The module I/O-IMC model. Input = {*recovered_B*, *recovering_2*}, Output = {*failed_B*, *up_B*}.

followed by *RU 2* outputting its 'failed' signal, then signal 'up_B' is received from module *B*, and finally *RU 2* outputs its own 'up' signal.

B.1.3 The recovery interface I/O-IMC

Figure B.4 shows the I/O-IMC model of *RU 2* recovery interface. The recovery interface receives a 'start_recover' signal from the RM (transition from state 0 to state 1), allowing it to start the RU's recovery. A 'recovering' signal is then output (transition from state 1 to state 2) notifying all the modules within the RU that a recovery phase has started (essentially disallowing any remaining operational module to fail). Then two sequential recoveries (i.e. of *B* and *C*) take place both with rate 1 (transitions from state 2 to state 3 and from state 3 to state 4), followed by two sequential 'recovered' notifications (transitions from state 4 to state 5 and from state 5 to state 0).

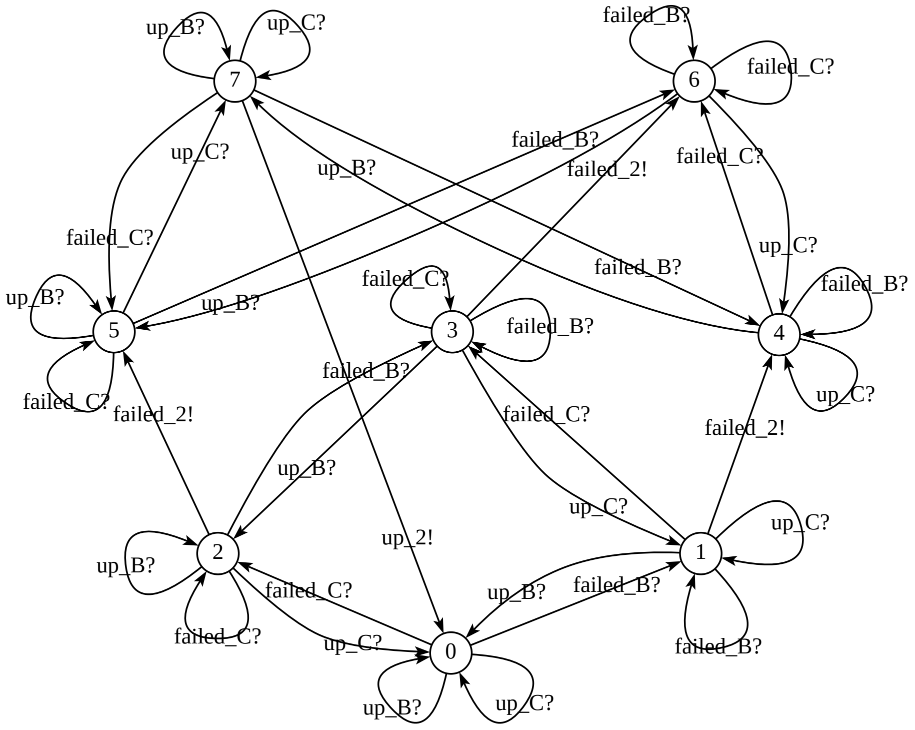


Figure B.3: The failure interface I/O-IMC model. Input = {*failed_B*, *up_B*, *failed_C*, *up_C*}, Output = {*failed_2*, *up_2*}.

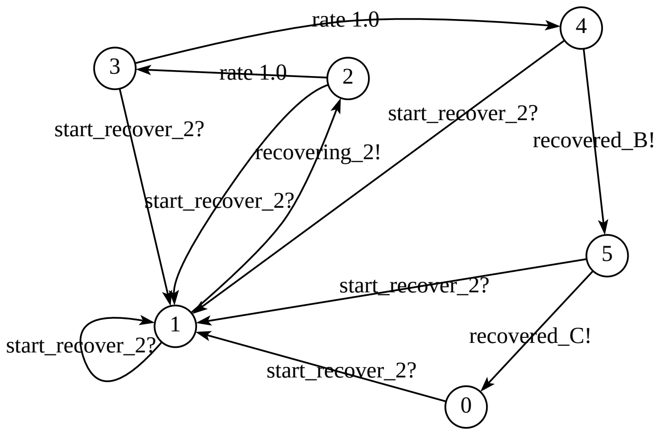


Figure B.4: The recovery interface I/O-IMC model. Input = {*start_recover_2*}, Output = {*recovering_2*, *recovered_B*, *recovered_C*}.

B.1.4 The recovery manager I/O-IMC

Figure B.5 shows the I/O-IMC model of the RM. The RM monitors the failure of *RU 1* and *RU 2*, and when an RU failure is detected, the RM grants its recovery by outputting a 'start_recover' signal. The RM has internally a queue of failing RUs that keeps track of the order in which the RUs have failed. The recovery policy of RM is to grant a 'start_recover' signal to the first failing RU. In queuing theory literature, this is referred to as a first-come-first-served (FCFS) policy. For instance, consider the following sequence of states: 0, 1, 4, 7, 2, 6, and 0; this corresponds to both RUs being initially operational, then *RU 1* fails, immediately followed by an *RU 2* failure. Since *RU 1* failed first, it is granted the 'start_recover' signal (transition from state 4 to state 7).Then, the RM waits for the 'up' signal of *RU 1*, and once received, RM grants the 'start_recover' signal to *RU 2* since *RU 2* is still in the queue of failing RUs (transition from state 2 to state 6). Finally, the RM receives 'up_2' and both RUs are operational again.

Note that any of the 4 I/O-IMC models presented here can be, to a certain extent, locally modified without affecting the remaining models. As an example, one might modify the RM by implementing a different recovery policy.

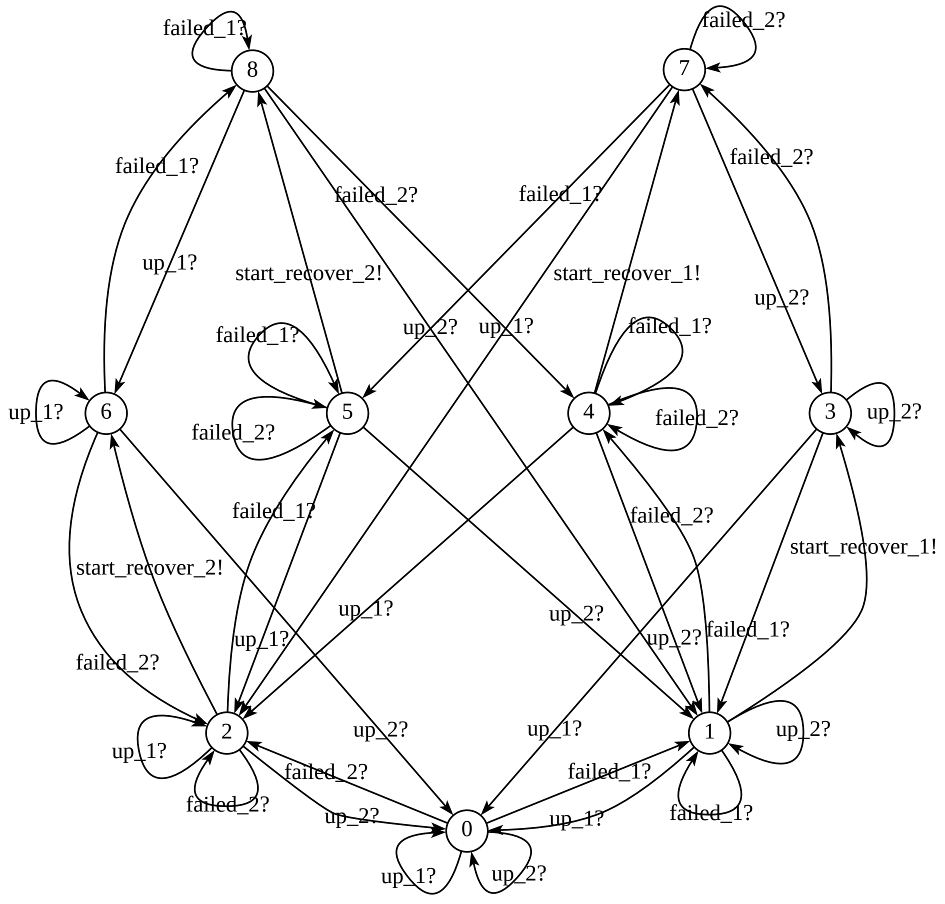


Figure B.5: The recovery manager I/O-IMC model. Input = {*failed_1*, *up_1*, *failed_2*, *up_2*}, Output = {*start_recover_1*, *start_recover_2*.}

B.2 I/O-IMC Model Specification with MIOA

We use a formal language called MIOA [71] to describe I/O-IMC models. MIOA is based on the IOA language defined by N. Lynch et al. in [45] and for which, in addition to interactive transitions, Markovian transitions have been added. The MIOA language is used for describing concisely and formally an I/O-IMC. It provides programming language constructs, such as control structures and data types, to describe complex system model behaviors. A MIOA specification is structured as shown in Figure B.6 and it is divided into 3 sections: *i*) *signature ii*) *states*, and *iii*) *transitions*. In the signature section, *input actions*, *output actions*, *internal actions* and *Markovian rates* are specified[1]. The set of data types specified in the states section determines the states of the I/O-IMC. Possible transitions of the I/O-IMC are specified in a precondition-effect style. In order for the transition to take place, the precondition has to hold. A precondition can be any expression that can be evaluated either to true or false. Since I/O-IMCs are input-enabled, there is no precondition specified for input actions.

```
IOIMC: <name>
  signature:
    input: <action_1>? ... <action_m>?
    output: <action_1>! ... <action_n>!
    internal: <action_1> ... <action_l>
    markovian: <rate_1> ... <rate_k>
  states:
    <state definitions>
    ...
  transitions:
    input: <action_i>?
    effect:
      ...
    output: <action_i>!
    precondition: <condition>
    effect:
      ...
    internal: <action_i>
    precondition: <condition>
    effect:
      ...
    markovian: <rate_i>
    precondition: <condition>
    effect:
      ...
```

Figure B.6: Basic building blocks of a MIOA specification.

The MIOA specification of the failure interface I/O-IMC model is shown in Figure B.7 as an example. The signature consists of actions that correspond to the failed/up signals of the n modules belonging to the RU (Line 3), one output signal for the 'failed' signal of the RU and one output signal for the 'up' signal of the RU (Line 4). The states of the failure interface I/O-IMC are defined (Lines 5-7) using *Set* and *Bool* data types, where 'set' (of size n) holds the names of the modules that have failed and 'rufailed' indicates if the RU has or has not failed. The initial state is also defined in the *states* section; for instance, the failure interface initial state is composed of 'set' being empty (Line 6) and 'rufailed' being false (Line 7). There are four kind of possible transitions; for example, the last transition (Lines 21-24) indicates that an RU 'up' signal (up_RU!) is output if 'set' is empty (i.e. all modules

[1]MIOA *actions* correspond to I/O-IMC *signals*.

are operational) and the RU has indeed failed at some point (i.e. ‘rufailed’ = true), and the effect of the transition is to set ‘rufailed’ to false.

```
 1: IOIMC: failure interface
 2:   signature:
 3:     input: failed(n:Int)? up(n:Int)?
 4:     output: failed_RU! up_RU!
 5:   states:
 6:     set: Set[n:Int] := {}
 7:     rufailed: Bool := false
 8:   transitions:
 9:     input: failed(i)?
10:     effect:
11:       if i ¬∈ set
12:         add(i, set)
13:     input: up(i)?
14:     effect:
15:       if i ∈ set
16:         remove(i, set)
17:     output: failed_RU!
18:     precondition: set.size() > 0 ∧ rufailed = false
19:     effect:
20:       rufailed := true
21:     output: up_RU!
22:     precondition: set.size() = 0 ∧ rufailed = true
23:     effect:
24:       rufailed := false
```

Figure B.7: MIOA specification of the failure interface I/O-IMC model.

Once a MIOA specification/description of an I/O-IMC model has been laid down, an algorithm can explore the state-space and generate the corresponding I/O-IMC model (see Section B.3 for details). In fact, automatically deriving the I/O-IMC models becomes essential as the models grow in size. For instance, the RM I/O-IMC that coordinates 7 RUs has 27,399 states and 397,285 transitions. In our modeling approach, the RM I/O-IMC size depends on the number of RUs, the failure/recovery interface I/O-IMC size depends on the number of modules within the RU, and the module I/O-IMC size is constant.

B.3 I/O-IMC Model Generation

We have implemented the I/O-IMC model generation based on the MIOA specifications. Algorithm 3 shows how the states and transitions of an I/O-IMC are generated based on its MIOA specification.

Algorithm 3 State space exploration and I/O-IMC generation based on MIOA specification

```
1: $stateSet \leftarrow \{\}$
2: $statesToBeProcessed \leftarrow \{\}$
3: $transitionSet \leftarrow \{\}$
4: $s \leftarrow intialize\ \mathit{mioa.states}$
5: $stateSet.add(s)$
6: $statesToBeProcessed.add(s)$
7: while $statesToBeProcessed \neq \{\}$ do
8:    $s \leftarrow statesToBeProcessed.removeLast()$
9:    for each $\mathit{mioa.transition}\ t$ do
10:      if $s.check(t.\mathit{precondition}) = \mathrm{TRUE}$ then
11:         $snew \leftarrow s.apply(t.\mathit{effect})$
12:         if $snew \notin stateSet$ then
13:            $stateSet.add(snew)$
14:            $statesToBeProcessed.add(snew)$
15:         end if
16:         $transitionSet.add(s.id, snew.id, t.\mathit{signal})$
17:      else
18:         if $t.\mathit{signal} = input$ then
19:            $transitionSet.add(s.id, s.id, t.\mathit{signal})$
20:         end if
21:      end if
22:   end for
23: end while
```

The algorithm keeps track of the set of states, the states that are yet to be evaluated and the set of transitions generated, which are all initialized as empty sets (Lines 1-3). An initial state is created which comprises the initialized *state variables* as defined in the MIOA specification (Line 4). The initial state is added to the state set and the set of states to be processed (Lines 5-6). Then, the algorithm iterates over the states in the set of states to be processed until there is no state left to be processed (Lines 7-8). For each state, all the transitions that are specified in the MIOA specification are evaluated (Line 9). If the *precondition* of the transition holds, a new state is created, on which the *effects* of the transition are reflected (Lines 10-11). If the resulting state does not already exist, it is added to the set of

```
 1: "X.bcg" = branching stochastic reduction of(
 2: (hide start_recover_1 in
 3:   (hide failed_B,up_B,failed_C,up_C in
 4: "fint_1.aut"
 5:     |[failed_1,failed_B,up_B,failed_C,up_C]|
 6:     (hide recovered_B in "module_B.aut"
 7:     |[recovered_B]|
 8:     (hide recovered_C in "module_C.aut"
 9:     |[recovered_C]|
10:     "rint_1.aut"))
11:   )
12:   |[start_recover_1,failed_1,up_1]|
13: (hide start_recover_2 in
14:   (hide failed_A,up_A in "fint_2.aut"
15:     |[failed_2,failed_A,up_A]|
16:     (hide recovered_A in "module_A.aut"
17:     |[recovered_A]|
18:     "rint_2.aut")
19:   )
20:   |[start_recover_2,failed_2,up_2]|
21: "recmgr.aut")));
```

Figure B.8: SVL script that composes the I/O-IMC models according to the example decomposition as shown in Figure B.1

states and the set of states to be processed (Lines 12-15). Also, a new transition from the original state to the resulting state is added to the set of transitions (Line 16). If the *precondition* of the transition does not hold for an *input signal*, then a self transition is added to the set of transitions to ensure that the generated I/O-IMC is *input-enabled* (Lines 17-22).

B.4 Composition and Analysis Script Generation

All the generated I/O-IMC models are output in the Aldebaran *.aut* file format so that they can be processed with the *CADP* tool-set [42]. In addition to all the necessary I/O-IMCs, a composition and analysis script is also generated, which conforms to the CADP SVL scripting language. Figure B.8 shows a part of the generated SVL script, which composes the I/O-IMC models for the example decomposition with 3 modules and 2 RUs (See Figure B.1).

The generated composition script first composes the recovery interface I/O-IMCs of RUs with the module I/O-IMCs that are comprised by the corresponding RUs. For instance, in Figure B.8, the I/O-IMCs of modules *B* and *C* are composed with the recovery interface I/O-IMC of *RU 1* (Lines 6-10). Similarly, the module *A* I/O-IMC is composed with the recovery interface of *RU 2* (Lines 16-18). The resulting I/O-IMCs are then composed with the failure interface I/O-IMCs of the corresponding RUs. Finally, all the resulting I/O-IMCs are composed with the RM I/O-IMC. At each composition step, the common input/output actions that are only relevant for the I/O-IMCs being composed are "hidden". That is, these actions are used for the composition of I/O-IMCs and eliminated in the resulting I/O-IMC. The execution of the generated SVL script within CADP composes and aggregates all the I/O-IMCs based on the modules decomposition, reduces the final I/O-IMC into a CTMC, and computes the steady-state availability.

Bibliography

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] G. Arrango. Domain analysis methods. In Schafer, R. Prieto-Diaz, and M. Matsumoto, editors, *Software Reusability*, pages 17–49. Ellis Horwood, 1994.

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[4] F. Bachman, L. Bass, and M. Klein. Deriving Architectural Tactics: A Step Toward Methodical Architectural Design. Technical Report CMU/SEI-2003-TR-004, SEI, Pittsburgh, PA, USA, 2003.

[5] L. Bas, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[6] W. Beaton and J. des Rivieres. Eclipse platform technical overview. Technical report, IBM, 2006. http://www.eclipse.org/articles/.

[7] P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet. Architecture-level modifiability analysis (ALMA). *Journal of Systems Software*, 69(1-2):129–147, 2004.

[8] P.O. Bengtsson and J. Bosch. Architecture level prediction of software maintenance. In *Proceedings of the Third European Conference Software Maintenance and Reengineering (CSMR)*, pages 139–147, Amsterdam, The Netherlands, 1999.

[9] D. Binkley. Source code analysis: A road map. In *Proceedings of the Conference on Future of Software Engineering (FOSE)*, pages 104–119, Washington, DC, USA, 2007.

[10] F. Boudali, P. Crouzen, and M. Stoelinga. A compositional semantics for dynamic fault trees in terms of interactive markov chains. In *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 4762 of *Lecture Notes in Computer Science*, pages 441–456, Tokyo, Japan, 2007. Springer-Verlag.

[11] H. Boudali, P. Crouzen, B. R. Haverkort, M. Kuntz, and M.I.A. Stoelinga. Architectural dependability evaluation with arcade. In *Proceedings of the 38th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 512–521, Anchorage, Alaska, USA, 2008.

[12] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, A System of Patterns*. Wiley, 1996.

[13] R. Buskens and O.J. Gonzalez. Model-centric development of highly available software systems. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems IV*, volume 4615 of *Lecture Notes in Computer Science*, pages 409–433. Springer-Verlag, 2007.

[14] D.G. Cacuci. *Sensitivity and Uncertainty Analysis: Theory*, volume 1. Chapman & Hall, 2003.

[15] G. Candea, J. Cutler, and A. Fox. Improving availability with recursive microreboots: A soft-state system case study. *Performance Evaluation*, 56(1-4):213–248, 2004.

[16] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot: A technique for cheap recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–44, San Francisco, CA, USA, 2004.

[17] K.M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[18] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.

[19] P. Clements, R. Kazman, and M. Klein. *Evaluating software architectures: methods and case studies*. Addison-Wesley, 2002.

[20] B. Cornelissen. Dynamic analysis techniques for the reconstruction of architectural views. In *Proceedings of the 14th Working Conference on Reverse*

Engineering (WCRE), pages 281–284, Vancouver, BC, Canada, 2007. IEEE Computer Society.

[21] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[22] O. Das and C.M. Woodside. The fault-tolerant layered queueing network model for performability of distributed systems. In *Proceedings of the International Performance and Dependability Symposium (IPDS)*, pages 132–141, Durham, NC, USA, 1998.

[23] E.M. Dashofy, A. van der Hoek, and R.N. Taylor. An infrastructure for the rapid development of XML-based architecture description languages. In *Proceedings of the 22rd International Conference on Software Engineering (ICSE)*, pages 266–276, Orlando, FL, USA, 2002. ACM.

[24] E.M. Dashofy, A. van der Hoek, and R.N. Taylor. Towards architecture-based self-healing systems. In *Proceedings of the first workshop on Self-healing systems (WOSS)*, pages 21–26, Charleston, SC, USA, 2002.

[25] P.A. de Castro Guerra, C.M.F. Rubira, A. Romanovsky, and R. de Lemos. A dependable architecture for cots-based software systems using protective wrappers. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*, volume 2667, pages 144–166. Springer-Verlag, 2003.

[26] V. de Florio and C. Blondia. A survey of linguistic structures for application-level fault tolerance. *ACM Computing Surveys*, 40(2):1–37, 2008.

[27] R. de Lemos, P. Guerra, and C. Rubira. A fault-tolerant architectural approach for dependable systems. *IEEE Software*, 23(2):80–87, 2004.

[28] I. de Visser. *Analyzing User Perceived Failure Severity in Consumer Electronics Products*. PhD thesis, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 2008.

[29] G. Deconinck, J. Vounckx, R. Lauwereins, and J.A. Peperstraete. Survey of backward error recovery techniques for multicomputers based on checkpointing and rollback. In *Proceedings of the IASTED International Conference on Modeling and Simulation*, pages 262–265, Pittsburgh, PA, USA, 1993.

[30] Department of Defense. Military standard: Procedures for performing a failure modes, effects and criticality analysis. Standard MIL-STD-1629A, Department of Defense, Washington DC, USA, 1984.

[31] L. Dobrica and E. Niemela. A survey on software architecture analysis methods. *IEEE Transactions on Software Engineering*, 28(7):638–654, 2002.

[32] J. Dollimore, T. Kindberg, and G. Coulouris. *Distributed Systems: Concepts and Design*. Addison Wesley, 2005.

[33] J.C. Duenas, W.L. de Oliveira, and J.A. de la Puente. A software architecture evaluation model. In *Proceedings of the Second International ESPRIT ARES Workshop*, pages 148–157, Las Palmas de Gran Canaria, Spain, 1998.

[34] J.B. Dugan. Software system analysis using fault trees. In M.R. Lyu, editor, *Handbook of Software Reliability Engineering*, chapter 15, pages 615–659. McGraw-Hill, 1996.

[35] J.B. Dugan and M.R. Lyu. Dependability modeling for fault-tolerant software and systems. In M. R. Lyu, editor, *Software Fault Tolerance*, chapter 5, pages 109–138. John Wiley & Sons, 1995.

[36] M. Elnozahy, L. Alvisi, Y. Wang, and D.B. Johnson. A survey of rollback-recovery protocols in message passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[37] T. Elrad, R. Fillman, and A. Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10):29–32, 2001.

[38] Erlang/OTP design principles, 2009. http://www.erlang.org/doc/.

[39] C.F. Eubanks, S. Kmenta, and K. Ishil. Advanced failure modes and effects analysis using behavior modeling. In *Proceedings of the ASME Design Theory and Methodology Conference (DETC)*, number DTM-02 in 97-DETC, Sacramento, CA, USA, 1997.

[40] J. Fenlason and R. Stallman. *GNU gprof: the GNU profiler*. Free Software Foundation, 2000. http://www.gnu.org/.

[41] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Design*. Addison-Wesley, 1995.

[42] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2006: A toolbox for the construction and analysis of distributed processes. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 158–163, Berlin, Germany, 2007. Springer-Verlag.

[43] D. Garlan, R. Allen, and J. OckerBloom. Architectural mismatch: why reuse is so hard. *IEEE Software*, 12:17–26, 1995.

[44] D. Garlan, S. Cheng, and B. Schmerl. Increasing system dependability through architecture-based self-repair. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*, pages 61–89. Springer-Verlag, 2003.

[45] S. Garland, N. Lynch, J. Tauber, and M. Vaziri. IOA user guide and reference manual. Technical Report MIT-LCS-TR-961, MIT CSAI Laboratory, Cambridge, MA, USA, 2004.

[46] S. Gokhale. Architecture-based software reliability analysis: Overview and limitations. *IEEE Transactions on Dependable and Secure Computing*, 4(1):32–40, 2007.

[47] A. Gorbenko, V. Kharchenko, and O. Tarasyuk. FMEA- technique of web services analysis and dependability ensuring. In M. Butler, C. Jones, A. Romanovsky, and E. Troubitsyna, editors, *Rigorous Development of Complex Fault-Tolerant Systems*, volume 4157 of *Lecture Notes in Computer Science*, pages 153–167. Springer-Verlag, 2006.

[48] K. Goseva-Popstojanova and K.S. Trivedi. Architecture-based approach to reliability assessment of software systems. *Performance Evaluation*, 45(2-3):179–204, 2001.

[49] V. Grassi, R. Mirandola, and A. Sabetta. An XML-based language to support performance and reliability modeling and analysis in software architectures. In R. Reussner, J. Mayer, J.A. Stafford, S. Overhage, S. Becker, and P.J. Schroeder, editors, *QoSA/SOQUA*, volume 3712 of *Lecture Notes in Computer Science*, pages 71–87. Springer-Verlag, 2005.

[50] J.M. Harris, J.L. Hirst, and M.J. Mossinghoff. *Combinatorics and Graph Theory*. Springer-Verlag, 2000.

[51] B.R. Haverkort and K.S. Trivedi. Specification techniques for markov reward models. *Discrete Event Dynamic Systems*, 3(2-3):219–247, 2006.

[52] A. Heddaya and A. Helal. Reliability, availability, dependability and performability: A user-centered view. Technical Report BU-CS-97-011, Boston University, 1997.

[53] J.N. Herder, H. Bos, B. Gras, P. Homburg, and A.S. Tanenbaum. Failure resilience for device drivers. In *Proceedings of the 37th Annual IEEE/IFIP*

International Conference on Dependable Systems and Networks (DSN), pages 41–50, Edinburgh, UK, 2007.

[54] H. Hermanns. *Interactive Markov Chains*, volume 2428 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[55] M. Hind. Pointer analysis havent we solved this problem yet? In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 54–61, Snowbird, UT, USA, 2001.

[56] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley, 1999.

[57] R.A. Howard. *Dynamic probability systems. Volume 1: Markov models*. John Wiley & Sons, 1971.

[58] Y. Huang and C. Kintala. Software fault tolerance in the application layer. In M. R. Lyu, editor, *Software Fault Tolerance*, chapter 10, pages 231–248. John Wiley & Sons, 1995.

[59] G.C. Hunt, M. Aiken, M. Fhndrich, C. Hawblitzel, O. Hodson, J. Larus, S. Levi, B. Steensgaard, D. Tarditi, and T. Wobber. Sealing OS processes to improve dependability and safety. *SIGOPS Operating Systems Review*, 41(3):341–354, 2007.

[60] IEEE. IEEE standard glossary of software engineering terminology. Standard 610.12-1990, IEEE, 1990.

[61] A. Immonen and E. Niemelä. Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and System Modeling*, 7(1):49–65, 2008.

[62] U. Isaksen, J.P. Bowen, and N. Nissanke. System and software safety in critical systems. Technical Report RUCS/97/TR/062/A, The University of Reading, UK, 1997.

[63] V. Issarny and J. Banatre. Architecture-based exception handling. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS)*, page 9058, Maui, Hawaii, USA, 2001.

[64] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[65] A. Jhumka, M. Hiller, and N. Suri. Component-based synthesis of dependable embedded software. In *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, pages 111–128, Oldenburg, Germany, 2002.

[66] K.C. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, SEI, 1990.

[67] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[68] S. Krishnan and D. Gannon. Checkpoint and restart for distributed components. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID)*, pages 281–288, Washington, DC, USA, 2004.

[69] P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[70] P. Kruchten. *The Rational Unified Process: An Introduction, Second Edition*. Addison-Wesley, 2000.

[71] G.W.M. Kuntz and B.R.H.M. Haverkort. Formal dependability engineering with MIOA. Technical Report TR-CTIT-08-39, University of Twente, 2008.

[72] C.D. Lai, M. Xie, K.L. Poh, Y.S. Dai, and P. Yang. A model for availability analysis of distributed software/hardware systems. *Information and Software Technology*, 44(6):343–350, 2002.

[73] M. Lanus, L. Yin, and K.S. Trivedi. Hierarchical composition and aggregation of state-based availability and performability models. *IEEE Transactions on Reliability*, 52(1):44–52, 2003.

[74] J.-C. Laprie, J. Arlat, C. Beounes, and K. Kanoun. Architectural issues in software fault tolerance. In M. R. Lyu, editor, *Software Fault Tolerance*, chapter 3, pages 47–80. John Wiley & Sons, 1995.

[75] N. Lassing, D. Rijsenbrij, and H. van Vliet. On software architecture analysis of flexibility, complexity of changes: Size isn't everything. In *Proceedings of the Second Nordic Software Architecture Workshop (NOSA)*, pages 1103–1581, Ronneby, Sweden, 1999.

[76] N.G. Leveson, S.S. Cha, and T.J. Shimeall. Safety verification of ada programs using software fault trees. *IEEE Software*, 8(4):48–59, 1991.

[77] C. Lung, S. Bot, K. Kalaichelvan, and R. Kazman. An approach to software architecture analysis for evolution and reusability. In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*, pages 144–154, Toronto, Canada, 1997.

[78] M.W. Maier, D. Emery, and R. Hilliard. Software architecture: Introducing IEEE Standard 1471. *IEEE Computer*, 34(4):107–109, 2001.

[79] I. Majzik and G. Huszerl. Towards dependability modeling of FT-CORBA architectures. In *Proceedings of the 4th European Dependable Computing Conference*, volume 2485 of *Lecture Notes in Computer Science*, pages 121–139. Springer-Verlag, 2002.

[80] I. Majzik, A. Pataricza, and A. Bondavalli. Stochastic dependability analysis of system architecture based on UML models. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*, volume 2667 of *Lecture Notes in Computer Science*, pages 219–244. Springer-Verlag, 2003.

[81] D.F. McAllister and M.A. Vouk. Fault-tolerant software reliability engineering. In M.R. Lyu, editor, *Handbook of Software Reliability Engineering*, chapter 14, pages 567–613. McGraw-Hill, 1996.

[82] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[83] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Eng.*, 32(3):193–208, 2006.

[84] G. Molter. Integrating saam in domain-centric and reuse-based development processes. In *Proceedings of the Second Nordic Workshop Software Architecture (NOSA)*, pages 1103–1581, Ronneby, Sweden, 1999.

[85] MPlayer official website, 2009. http://www.mplayerhq.hu/.

[86] N. Lynch and M. Tuttle. An Introduction to Input/output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

[87] M.L. Nelson. A survey of reverse engineering and program comprehension. *Computing Research Repository (CoRR)*, abs/cs/0503068, 2005. http://arxiv.org/abs/cs/0503068.

[88] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Notices*, 42(6):89–100, 2007.

[89] Object Management Group. Fault tolerant CORBA. Technical Report OMG Document formal/2001-09-29, Object Management Group, 2001.

[90] Y. Papadopoulos, D. Parker, and C. Grante. Automating the failure modes and effects analysis of safety critical systems. In *Proceedings of the 8th IEEE International Symposium on High Assurance Systems Engineering (HASE)*, pages 310–311, Tampa, FL, USA, 2004.

[91] E. Parchas and R. de Lemos. An architectural approach for improving availability in web services. In *Proceedings of the Workshop on Architecting Dependable Systems (WADS)*, pages 37–41, Edinburgh, UK, 2004.

[92] M. Pistoia, S. Chandra, S.J. Fink, and E. Yahav. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM Systems Journal*, 46(2):265–288, 2007.

[93] M. Rakic and N. Medvidovic. Increasing the confidence in off-the-shelf components: a software connector-based approach. *ACM SIGSOFT Software Engineering Notes*, 26(3):11–18, 2001.

[94] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, Los Angeles, CA, USA, 1975.

[95] B. Randell. Turing memorial lecture: Facing up to faults. *Computer Journal*, 43(2):95–106, 2000.

[96] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Colub, and M. Jones. Mach: A system software kernel. In *Proceedings of the 34th Computer Society International Conference (COMPCON)*, pages 176–178, San Francisco, CA, USA, 1989.

[97] F. Redmill. Exploring subjectivity in hazard analysis. *IEE Engineering Management Journal*, 12(3):139–144, 2002.

[98] D.J. Reifer. Software failure modes and effects analysis. *IEEE Transactions on Reliability*, R-28(3):247–249, 1979.

[99] E. Roland and B. Moriarty. Failure mode and effects analysis. In *System Safety Engineering and Management*. John Wiley & Sons, 1990.

[100] D. Rosenberg and K. Scott. *Use Case Driven Object Modeling with UML: A Practical Approach*. Addison-Wesley, 1999.

[101] S.M. Ross. *Introduction to Probability Models*. Elsevier Inc., 2007.

[102] A.-E. Rugina, K. Kanoun, and M. Kaniche. A system dependability modeling framework using AADL and GSPNs. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems IV*, volume 4615 of *Lecture Notes in Computer Science*, pages 14–38. Springer-Verlag, 2006.

[103] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[104] J. Rumbaugh, I. Jacobson, and G. Booch, editors. *The Unified Modeling Language reference manual*. Addison-Wesley, 1999.

[105] F. Ruskey. Simple combinatorial gray codes constructed by reversing sublists. In *Proceedings of the 4th International Symposium on Algorithms and Computation (ISAAC)*, volume 762 of *Lecture Notes in Computer Science*, pages 201–208. Springer-Verlag, 1993.

[106] F. Ruskey. *Combinatorial Generation*. University of Victoria, Victoria, BC, Canada, 2003. Manuscript CSC-425/520.

[107] T. Saridakis. Design patterns for graceful degradation. In *Proceedings of the 10th European Conference on Pattern Languages and Programs (EuroPloP)*, pages E7/1–18, Irsee, Germany, 2005.

[108] B. A. Schroeder. On-line monitoring: A tutorial. *IEEE Computer*, 28:72–78, 1995.

[109] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[110] H. Sozer and B. Tekinerdogan. Introducing recovery style for modeling and analyzing system recovery. In *Proceedings of the 7th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 167–176, Vancouver, BC, Canada, 2008.

[111] H. Sozer, B. Tekinerdogan, and M. Aksit. Extending failure modes and effects analysis approach for reliability analysis at the software architecture design level. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems IV*, volume 4615 of *Lecture Notes in Computer Science*, pages 409–433. Springer-Verlag, 2007.

[112] H. Sozer, B. Tekinerdogan, and M. Aksit. FLORA: A framework for decomposing software architecture to introduce local recovery. *Software: Practice and Experience*, 2009. Accepted.

[113] F. Tartanoglu, V. Issarny, A. Romanovsky, and N. Levy. Dependability in the web services architectures. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*, pages 90–109. Springer-Verlag, 2003.

[114] R.N. Taylor, N. Medvidovic, K.M. Anderson, Jr. E.J. Whitehead, J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering*, 22(6):390–406, 1996.

[115] B. Tekinerdogan. *Synthesis-Based Software Architecture Design*. PhD thesis, University of Twente, Enschede, The Netherlands, 2000.

[116] B. Tekinerdogan. Asaam: Aspectual software architecture analysis method. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 5–14, Oslo, Norway, 2004.

[117] B. Tekinerdogan, H. Sozer, and M. Aksit. Software architecture reliability analysis using failure scenarios. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 203–204, Pittsburgh, PA, USA, 2005.

[118] B. Tekinerdogan, H. Sozer, and M. Aksit. Software architecture reliability analysis using failure scenarios. *Journal of Systems and Software*, 81(4):558–575, 2008.

[119] R. Tischler, R. Schaufler, and C. Payne. Static analysis of programs as an aid to debugging. *SIGPLAN Notices*, 18(8):155–158, 1983.

[120] Trader project, ESI, 2009. http://www.esi.nl.

[121] K.S. Trivedi and M. Malhotra. Reliability and performability techniques and tools: A survey. In *In Proceedings of the 7th Conference on Measurement, Modeling, and Evaluation of Computer and Communication Systems (MMB)*, pages 27–48, Aachen, Germany, 1993.

[122] K. Vaidyanathan and K.S. Trivedi. A comprehensive model for software rejuvenation. *IEEE Transactions on Dependable and Secure Computing*, 2(2):124–137, 2005.

[123] R.F. van der Lans. *The SQL Standard: a Complete Guide Reference*. Prentice Hall, 1989.

[124] M. Wallace. Modular architectural representation and analysis of fault propagation and transformation. In *Proceedings of the Second International Workshop on Formal Foundations of Embedded Software and Component-based Software Architectures (FESCA)*, pages 53–71, Edinburgh, UK, 2005.

[125] E. Woods and N. Rozanski. Using architectural perspectives. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 25–35, Pittsburgh, PA, USA, 2005.

[126] S. Yacoub, B. Cukic, and H.H. Ammar. A scenario-based reliability analysis approach for component-based software. *IEEE Transactions on Reliability*, 53(14):465–480, 2004.

[127] Q. Yang, J.J. Li, and D.M. Weiss. A survey of coverage based testing tools. In *Proceedings of the International Workshop on Automation of Software Test (AST)*, pages 99–103, Shanghai, China, 2006.

[128] J. Zhou and T. Stalhane. Using FMEA for early robustness analysis of web-based systems. In *Proceedings of the 28th International Computer Software and Applications Conference (COMPSAC)*, pages 28–29, Hong Kong, China, 2004.

Samenvatting

De toenemende grootte en de complexiteit van softwaresystemen vermoeilijken dat alle mogelijke fouten kunnen worden verhinderd dan wel tijdig worden verwijderd. De fouten die in het systeem blijven bestaan kunnen er uiteindelijk toe leiden dat het gehele systeem faalt. Teneinde hiermee om te kunnen gaan zijn er in de literatuur verscheidene zogenaamde fouten-tolerantietechnieken geïntroduceerd. Deze technieken gaan ervan uit dat niet alle fouten verhinderd kunnen worden, en staan daarom toleranter tegenover mogelijke fouten in het systeem. Daarbij zorgen deze technieken ervoor dat het systeem zich kan herstellen indien fouten in het systeem zijn gedetecteerd. Hoewel deze fouten-tolerantietechnieken zeer bruikbaar kunnen zijn is het echter niet altijd eenvoudig ze in het systeem te integreren. In dit proefschrift richten we onze aandacht op de volgende problemen bij het ontwerpen van fouten-tolerante systemen.

Ten eerste behandelen de huidige fouten-tolerantietechnieken niet expliciet fouten die door de eindgebruiker waarneembaar zijn. Ten tweede zijn bestaande architectuur stijlen niet geschikt voor het specificeren, communiceren en analyseren van ontwerpbeslissingen die direct zijn gerelateerd aan fouten tolerantie. Ten derde zijn er geen adequate analyse technieken die de impact van de introductie van fouten-tolerantie technieken op de functionele decompositie van de software architectuur kunnen meten. Ten slotte, vereist het grote inspanning om een fouten-tolerante ontwerp te realiseren en te onderhouden.

Om het eerste probleem aan te pakken introduceren we de methode SARAH die gebaseerd is op scenario-gebaseerde analyse technieken en de bestaande betrouwbaarheidstechnieken (FMEA en FTA) ten einde een software architectuur te analyseren met betrekking tot betrouwbaarheid. Naast het integreren van architectuur analyse methoden met bestaande betrouwbaarheidstechnieken, richt SARAH zich uitdrukkelijk op het identificeren van mogelijke fouten die waarneembaar zijn door de eindgebruiker, en hiermee het identificeren van gevoelige punten, zonder hierbij een implementatie van het systeem te vereisen.

Voor het specificeren van fouten-tolerantie aspecten van de software architectuur introduceren wij de zogenaamde Recovery Style. Deze wordt gebruikt voor het communiceren en analyseren van architectuurontwerpbeslissingen en voor het ondersteunen van gedetailleerd ontwerp met betrekking tot het herstellen van het systeem.

Voor het oplossen van het derde probleem introduceren we een systematische methode voor het optimaliseren van de software architectuur voor lokale herstelling van fouten (local recovery), opdat de beschikbaarheid van het systeem wordt verruimd. Om de methode te ondersteunen, hebben wij een gentegreerd reeks van gereedschappen ontwikkeld die gebruik maken van optimalisatietechnieken, toestandsgebaseerde analytische modellen (CTMCs) en dynamische analyse van het systeem. De methode ondersteunt de volgende aspecten: modelleren van de ontwerpruimte van de mogelijke architectuur decompositie alternatieven, het reduceren van de ontwerpruimte op basis van de domein en stakeholder constraints, en het balanceren en selectie van de alternatieven met betrekking tot de beschikbaarheid en prestatie (performance) metrieken.

Om de inspanningen voor de ontwikkeling en onderhoud van fouten-tolerante aspecten te reduceren is er een framework, FLORA, ontwikkeld die de decompositie en de implementatie van softwarearchitectuur voor lokale herstelling ondersteunt. Het framework verschaft herbruikbare abstracties voor het definiëren van herstelbare eenheden en voor het integreren van de noodzakelijke coördinatie en communicatie protocollen die vereist zijn voor herstelling van fouten.

Index

Titles in the IPA Dissertation Series since 2002

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications.* Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications.* Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems.* Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs.* Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell.* Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation.* Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets.* Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics.* Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station.* Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents.* Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures.* Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra.* Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-*

Reactive Scheduling. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems.* Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication.* Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell.* Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model.* Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems.* Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems.* Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality.* Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications.* Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting.* Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs.* Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations.* Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured*

Coverage. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation.* Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains.* Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use.* Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems.* Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML.* Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery.* Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding.* Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks.* Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data.* Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy.* Faculty of Science, Mathematics and Computer Science, RU. 2008-01