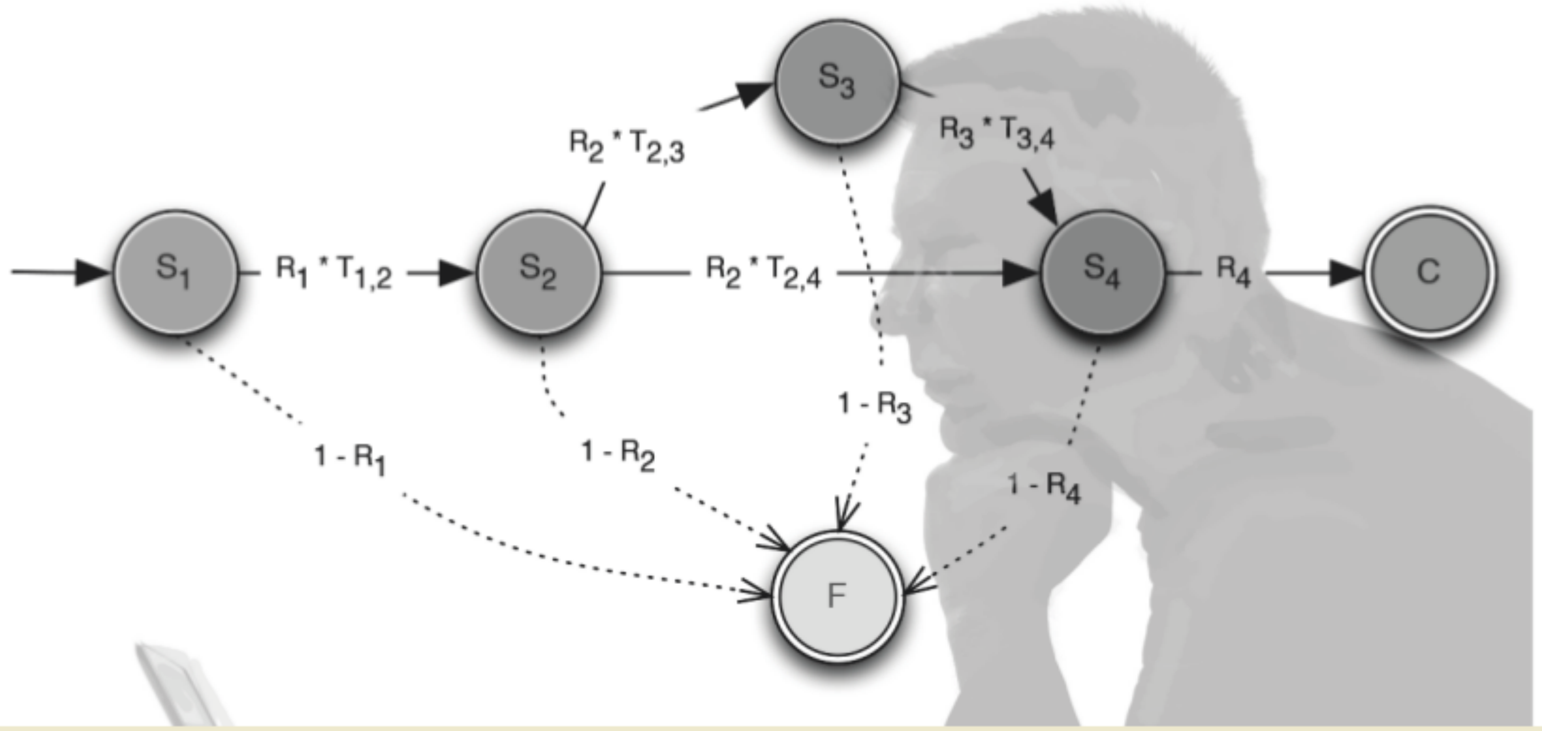


João Miguel da Costa Sousa Franco

Automated Reliability Prediction and Analysis from Software Architectures

Doctoral Program in Information Science and Technology, supervised by Mário Rela and Raul Barbosa and presented at the Department of Informatics Engineering of the Faculty of Science and Technology of the University of Coimbra

September 2015

· U C ·

Universidade de Coimbra

Automated Reliability Prediction and Analysis from Software Architectures

João Miguel Costa Sousa Franco

Thesis submitted to the University of Coimbra
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Information Science and Technology

September 2015

Department of Informatics Engineering
Faculty of Science and Technology

University of Coimbra

This research has been developed as part of the requirements of the Doctoral Program in Information Science and Technology of the Faculty of Sciences and Technology of the University of Coimbra. This work was carried out in the Software and Systems Engineering Group of the Center for Informatics and Systems of the University of Coimbra (CISUC).

This work has been supervised by **Professor Mário Zenha Rela** and **Professor Raul Barbosa** at the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

Funding for this work was partially provided by the Portuguese Research Agency *Fundação para a Ciência e Tecnologia* (FCT) through the scholarship SFRH/BD/89702/2012.

FCT
Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA EDUCAÇÃO E CIÊNCIA

POPH

Abstract

The quality of a software is determined by how it meets non-functional requirements such as performance, reliability, availability, maintainability and other '-ilities'. Depending on the application context, certain qualities are more critical to attain than others. As an example, a web-server processing large amounts of data should present qualities regarding to performance, while a software applied to a medical context must assure that no human life is at risk and as such, should comply to safety as a quality requirement.

In a software engineering perspective, quality requirements should be assessed throughout the software development life-cycle. In an early stage, quality assessment supports design decisions and promotes analysis of possible alternatives. During the implementation or testing stages, project managers may confirm that the developed product complies with the design and assure that it will conform to the stakeholder's requirements. Regarding evolutionary stages, architects can also compare different designs and decide for the most suitable solution taking into account the desired quality attributes.

During the development of a software system, neglecting the assessment of the quality requirements may lead, sooner or later, to the developed product failing to achieve one or more non-functional attributes desired by the stakeholder. Consequently, the development process returns to a previous phase for re-designing, re-implement and re-test a new solution to solve the problem. In short, it will involve more time, effort and money, causing more costs to the whole software project.

Software architecture plays an important role in the achievement of non-functional attributes. Designers use architectures to reason about non-functional properties and maintain their traceability through the project lifespan. Software architectures allow to structure the system in early stages of the development by describing implementation plans and specify rules, properties and architectural styles to attain specific quality attributes. For these reasons, the existing techniques to assess quality attributes use software architectures to obtain information about the system and provide accurate quantitative results.

The problem addressed by this thesis resides in the fact that most of the methods currently used to assess quality attributes from a software architecture are still performed manually. To quantitatively assess an architecture's quality attribute, designers have to build mathematical models through manual tasks and rebuild them for every change performed in the architecture. As any other manual task, building such models is error-prone, time consuming and is unfeasible for large and complex systems.

With this in mind, this thesis proposes to fill a gap in research by investigating towards a method that automatically assesses the reliability as a quality attribute from a software architecture. In particular, we exploit the formalisms of Architecture Description Languages (ADLs) to automatically generate mathematical models expressing the reliability behavior of a system. Then, we extended the notion of 'automated assessment' to perform a thorough analysis to identify architectural weak points that are affecting the system. This analysis aims to provide information for architects about reliability improvements and suggest alternatives.

With the goal of providing an assisting tool to aid architects in the design process, we implemented a plugin integrated in a ADL design tool. This plugin aims to make our automated approach available for architects to test and analyze their designs regarding reliability. In addition, we showed the different application contexts of our approach

by including it in the reasoning process of self-adaptive systems. The results showed an improvement in the overall system quality when comparing to the traditional planning approaches.

To conclude, we validated our method through a set of experiments that put into comparison our method with others that used manual approaches to assess reliability. In this work we pursue the motivation of contributing with a set of methods to give support for practitioners and researchers to avoid, prevent and detect undesired or unfeasible architectural designs. Moreover, we intend to promote the development of software with better quality and assure that it meets the desired quality requirements during the development process.

Resumo

A qualidade do *software* é determinada a partir da forma como os requisitos não-funcionais são alcançados, tais como o desempenho, a fiabilidade, a disponibilidade, a manutenção, entre outros. Dependendo do contexto aplicacional, certos atributos de qualidade são mais críticos de alcançar do que outros. Por exemplo, um servidor *web* processa grandes quantidades de informação pelo que deve apresentar atributos de qualidade que respondam ao desempenho requerido, já o *software* aplicado a um contexto médico deve assegurar que nenhuma vida humana é colocada em risco e garantir a integridade física dos utilizadores.

Na perspectiva da engenharia de *software*, os requisitos de qualidade devem ser avaliados e testados ao longo do processo de desenvolvimento do *software*. Numa fase inicial, esta avaliação permite apoiar as decisões de *design* e promover a análise de possíveis alternativas. Durante a implementação e a fase de testes, os gestores de projecto poderão confirmar que o *design* inicial corresponde ao produto desenvolvido e assegurar que cumpre os requisitos do cliente. Em relação à fase evolutiva, poderão, ainda, comparar diferentes *designs* e decidir sobre a melhor solução disponível no que concerne aos objectivos de qualidade.

Durante o desenvolvimento de um *software*, negligenciar a avaliação dos atributos de qualidade pode, mais tarde ou mais cedo, levar a que o produto falhe por não satisfazer os objectivos de qualidade requeridos pelo cliente. Consequentemente, o processo de desenvolvimento terá de voltar a um estado prévio de modo a conceber uma nova solução, reimplementar um novo desenho e voltar a testar o produto final.

Este processo acabará por requerer mais tempo, esforço e dinheiro, tornando, inevitavelmente, todo o projecto mais oneroso.

A arquitectura desempenha um papel importante na obtenção dos atributos de qualidade, serve como base para os *designers* codificarem propriedades não-funcionais e empregarem boas práticas de *design*. Permite também manter um registo das alterações durante a vida do *software* e pode ainda ser utilizada como meio de comunicação entre clientes, programadores, *designers* e responsáveis pela manutenção do sistema. A arquitectura pode ser considerada como um dos primeiros documentos do projecto, já que permite estruturar o sistema, descrever os planos de desenvolvimento e especificar regras, propriedades e estilos arquitectónicos, de modo a alcançar atributos de qualidade específicos. Por estas razões, as técnicas existentes para avaliar os atributos de qualidade utilizam arquitecturas de *software* para obter informações acerca do sistema e fornecer resultados quantitativos precisos.

Esta tese pretende abordar a problemática existente no facto de, no mundo actual, grande parte dos métodos que avaliam os atributos de qualidade das arquitecturas ainda serem realizados manualmente. Para avaliar quantitativamente um atributo de qualidade, os *designers* têm de construir modelos matemáticos através de tarefas manuais e reformulá-los sempre que se registar uma mudança no sistema. Como em qualquer outra tarefa manual, a construção dos ditos modelos é susceptível a erros, morosa e pode mesmo ser inviável para sistemas complexos e de larga-escala.

Assim, considerando, o supra referido, esta tese propõe-se preencher uma lacuna nos métodos actuais de avaliação, investigando um método automático que avalie a fiabilidade enquanto atributo de qualidade de uma arquitectura de *software*. Pretendemos, em particular, explorar os formalismos de *Architectural Description Languages* (ADLs) de forma a gerar modelos matemáticos que expressem o comportamento da fiabilidade de um sistema. Numa fase posterior, implementaremos

uma análise exaustiva no sistema que permita identificar pontos fracos a nível arquitectónico. Esta análise pretende, pois disponibilizar informação sobre melhorias na fiabilidade e sugerir alternativas aos arquitectos.

Com o objectivo de criar uma ferramenta que apoie os utilizadores no processo de *design*, implementámos um *plugin* integrado numa ferramenta de criação de ADLs. Esta abordagem pretende disponibilizar um método automático que permita aos arquitectos testar e analisar os seus *designs* relativamente à fiabilidade. Adicionalmente, alargámos o contexto da aplicabilidade da nossa técnica de avaliação automática, de modo a demonstrar a diversidade da sua utilização. Com este intuito aplicámo-la ao processo de planeamento dos sistemas auto-adaptativos, o que resultou numa melhoria da qualidade do sistema quando comparado às abordagens tradicionais.

Em conclusão, validámos a nossa linha de investigação através de um conjunto de experiências que comparam o resultado do nosso método automático com métodos manuais de avaliação da fiabilidade. Neste trabalho pretendemos contribuir com um conjunto de métodos que forneçam suporte aos utilizadores e investigadores de forma a evitar, prevenir e detectar *designs* arquitectónicos indesejáveis ou impraticáveis. Além disso, pretendemos ainda promover o desenvolvimento de *software* com maior qualidade e assegurar que este cumpre os objectivos requeridos ao longo do processo de desenvolvimento.

To

Armando and Aurora

My dear family

My beloved Joana

Acknowledgements

I deeply appreciate my advisors and mentors Mário Rela and Raul Barbosa. Mário for his patience, insight, encouragement, full support and inspiring daily smile. Raul for his guidance, motivation, teach, mountain biking, two-hours-lunches, sushi and fine taste of wine.
My gratitude to Francisco Correia for his commitment, hard-work and long discussions about self-adaptive systems (sometimes until long hours in the night). Vitor Silva for teaching me the pleasure of guiding and mentoring a young promising individual and Frederico Cerveira for his effort and exchange of ideas on virtualization of operating systems. My heartfelt thanks to David Garlan, Bradley Schmerl and Paulo Casanova for the feedback, suggestions, criticisms and all the hours spent on Skype and Hangout – an invaluable support.

Joana Pacheco
for her support, encouragement, great food, Timtim and her loving care!

"Grupo dos vossos amigos"
because a man cannot happily live without the companion, laughs, beers, 347 'moscatéis' and the tough 'preferias' challenges of his friends.

PhD colleagues
for distracting me and interrupt my work – always for a good cause (thanks for the oranges)

Ex-Housemates
for beers, conversations, laughs, dinners and games!

Quartel (12^{o}F)
for that crazy and amusing once a year dinner

Dognædis friends
for encouraging and supporting my PhD enrollment

SSE of CISUC
for the discussions, suggestions and inspiring developed works. Among these, a special and comforting thanks to the members of the Special Interest Group on Dependability (SIGDep)

Last, but certainly not least, my gratitude to all my sisters, Dona Teresa, my gramps, aunt, uncle and whole family for shaping me in the early years and giving the smile that ignited the required inspiration and motivation for this work. My eternal grace to my father and my mother for absolutely everything!!!

Contents

List of Figures

List of Tables

List of Abbreviations

AADL	Architecture Analysis and Design Language
ADL	Architecture Description Language
AFR	Annualized Failure Rate
AGREE	Advisory Group on Reliability of Electronic Equipment
CMU	Carnegie Mellon University
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
CSL	Continuous Stochastic Logic
CSP	Communicating Sequential Processes
CTL	Computation Tree Logic
CTMC	Continuous-Time Markov Chain
DNS	Domain Name System
DTMC	Discrete-Time Markov Chain
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
ID	Identification

IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization
LTL	Linear Temporal Logic
MDP	Markov Decision Process
MRMC	Markov Reward Model Checker
MTBF	Mean Time Between Failures
MTBM	Meant Time Between Maintenance
MTTF	Mean Time To Failure
MTTR	Mean Time To Repair
PAT	Process Analysis Toolkit
PCTL	Probabilistic Computation Tree Logic
PC	Personal Computer
RPC	Remote Procedure Call
SEI	Software Engineering Institute
TMR	Triple Modular Redundancy
VDM	Vienna Development Method

Chapter 1

Introduction

Users are becoming more dependent on software systems and expect them not to only deliver functional attributes, (e.g., using smartphones to browse the internet, listen to music), but to also present quality properties like performance, reliability, security and other '-ilities'. To achieve those, software systems must be designed and planned with quality goals[1] in mind, since early stages of the software development life-cycle influence the quality of the software.

Assessing quality goals in such early phases, allows designers to reflect on the system, test alternatives and compare new designs with the aim of finding one that best suits the stakeholders' needs. However, this creates major challenges for the development, maintenance and evolution of software. Since, its professionals struggle to master tools and methodologies that have not been designed to handle these qualities and may determine the success or the failure of large, modern and distributed systems. Moreover, one important quality attribute of such systems is reliability, due to its intrinsic meaning of failure-free operation. In other words, the systems are expected to behave correctly and deliver the expected services at all times, otherwise the system enters in an erroneous state.

Reliability assessment from a system in an early stage, before any implementation, requires specifying the system under design and employing a mathematical method to obtain a quantitative prediction of its reliability. The former, system specification, can be detailed through an architecture using an Architecture

[1]The terms 'quality goals' and 'non-functional attributes' are used interchangeably throughout this thesis.

Description Languages (ADLs). These ADLs allow designers to model, represent and describe a software architecture, thereby improving the artifacts used for communication between designers, developers and stakeholders. The latter, mathematical methods, vary greatly depending on the quality goal under assessment. In architecture-based reliability, a commonly accepted method is the use of stochastic processes to model the randomness caused by the failure process and interactions between the architectural elements. The driving-force behind the use of reliability prediction methods to assess a software description, includes the following:

- An assurance early in the design phase that the product meets the stakeholders' requirements;
- An analysis of to what extent the system reliability depends on architectural elements and interconnections;
- A comparison of the reliability of different architectural alternatives;
- A study of the sensitivity of the system reliability and suggestions for improvements;
- The adoption of approaches that will be applicable throughout the software life-cycle.

During the design phase of a software, reliability assessment provides assurances for architects about the design and whether it fulfills the requirements of the stakeholders. In this early phase, this assessment can be undertaken by using failure data obtained from reusable and Commercial Off-The-Shelf (COTS) components. If the architecture uses components that are developed in-house, the architects can estimate the failure rate data by contacting experienced developers and although on the edge of the unknown, they can still investigate the impact of the architectural elements and interconnections for a given architecture. The information provided from this kind of analysis allows project managers to redefine new designs and alternatives that can be readily compared or even test 'what-if' scenarios. What-if an architectural element is more or less reliable or a connection is more or less frequently used? Architecture-based reliability employs mathematical models that make it possible to study the sensitivity of the whole system reliability to variations within each architectural element. This allows the critical components or interconnections that affect the system reliability to be identified and improvements to be made which might lead to a higher overall system reliability.

Assessing reliability of a software architecture involves producing a mathematical model with a complexity that varies in proportion to the size of the system. In other words, large complex systems may yield large complex models. In modern computer systems with an increase of computational power and memory, processing a large amount of data is not a difficult task. However, current practice still relies on building these stochastic models by hand without any assisting tool or method to help in the generation process. Thus, generating these models manually is as prone to errors as any other manual task; in addition, it requires modeling expertise and is time-consuming because of the overwhelmingly complex designs. In view of this, an automated method to verify, validate and test reliability of a software architecture is becoming essential.

Therefore, this dissertation addresses an important research question: **how can reliability be quantitatively assessed and analyzed from a software description while avoiding time and effort consuming tasks?**. Such approach opens up a new avenue for an in-depth analysis which is not possible through manual means. Moreover, this analysis can be applied not only in the design phase, but also in later stages where the evolution of the current system is under consideration. In these phases, architects can plan modifications by drawing on information from the deployed system to test, analyze and compare design alternatives through an automated design assisting tool.

To address this research question, we developed a design assisting tool as a plugin for an Integrated Development Environment (IDE) framework. This framework allows software architectures to be described through ADL specifications and our tool provides architects with the means to test and analyze their designs. We aim to equip design frameworks with testing abilities that can allow architects to ensure the required quality goals are achieved and compare different architectural alternatives.

The research undertaken in this dissertation focuses on the automated assessment of quality attributes with an emphasis on reliability. The achievements and contributions made to the research community resulting from this dissertation are outlined in the following section.

1.1 Contributions

In this thesis we intend to bridge the gap between research and practitioner communities by providing the means for software designers and architects to build more dependable systems. The following comprises a detailed of what has been achieved in this area:

- The development of an automated method to generate stochastic models from software architectures which allows the reliability of a system to be predicted. The generation includes the use of different architectural styles which may display known quality attributes in particular conditions (Chapter 3);
- The description of a formal notation to simplify the translation process and bring about the automated generation of stochastic models from software architectures (Chapter 3).
- The application of a sensitivity analysis on the reliability of the system for each component and connector to identify bottlenecks and possible architectural issues that require attention (Chapter 4);
- The development of a tool that automatically assesses, predicts and analysis the reliability of a software architecture. This tool has been integrated within an architecture design framework to provide feedback about the quality of the designed artifact (Chapter 5);
- A validation of our approach by comparing the automated generation method with the manual assessment of other studies that applied to the same scenarios (Chapter 5);

- Support for the planning phase of self-adaptive systems. To this end, we predict the quality outcome for each adaptation strategy at runtime, by allowing the system to make more informed decisions (Chapter 6).
- The adoption of an approach to model the failure pathology of a software component by encompassing error masking, error propagation, multiple failure modes and failure recoveries (Chapter 7).
- Undertaking a case-study based on a widely used cloud infrastructure system to validate our model of failure pathology. Furthermore, the obtained failure rate data can be used in future studies as a basis to test, compare and validate different assessment methods (Chapter 7).

1.2 Structure of the Thesis

The **first chapter** introduces the research problem and addresses the main significant areas of this thesis.

Chapter 2 clarifies the main concepts of this thesis and answers questions such as "What is software architecture?" or even "How should we define reliability?". In addition, we explore the mathematical methods that can be used for assessing reliability from a software architecture.

Chapter 3 proposes an approach to predict the reliability of a software architecture in an automated fashion. In addition, we propose a formal notation to extend the proposed approach to other quantitative quality attributes, such as performance.

Chapter 4 sets out a method to analyze system reliability in an automated fashion. This method makes it possible to identify architectural bottlenecks and prioritize the components and connections that need urgent improvement.

Chapter 5 describes the implementation of the Affidavit tool which provides assessment and analysis capabilities for the architect, and is made accessible from an architectural design tool. Following this, we validate our approach by comparing the results of our automated method with those from the literature.

Chapter 6 examines in detail the application of the method outlined in Chapter 3 that gives support to the planning phase of a self-adaptive system. To this end, we employed our automated reliability prediction method to ensure the desired non-functional goals could be achieved in every performed adaptation, even in untested conditions.

Chapter 7 sets out a method to model reliability and proposes its automated prediction by extending the formal notation described in Chapter 3. We implemented a realistic experimentation scenario to validate this method and compare the obtained experimental results with those modeled from our approach.

Chapter 8 concludes this thesis by summing up the results of the research and recommending new avenues for future work.

List of Publications details the outcome of this thesis in terms of research publications in Conferences and Journals.

Chapter 2

Background

The assessment of reliability from a software architecture has been studied since the mid-1980s [Cheung, 1980] and since then, several new methods and techniques have been explored [Goševa-Popstojanova & Trivedi, 2001]. These methods serve as valuable research strategies with a great potential to support architects during the design and evolutionary phases of the software development life-cycle. However, current state-of-the-art approaches reveal a gap between the theory and practice due to the lack of tools and automated methods to assess reliability from an architectural description.

These state-of-the-art methods assess reliability in the design phase by building state-space models through manual means. Whenever a change in the architecture occurs, these models have to be rebuild, involving a considerable amount of effort and time. With this in mind, our work aims to provide the means for architects to test and analyze their designs. The adopted approaches require little effort and avoid the issues that often arise from manual activities: proneness to error and time-consuming tasks.

In this chapter we begin by explaining what a software architecture (2.1) is, its foundations, constituent features and their observable quality attributes. When certain qualities are attained, dependability (2.2) can be achieved, which provides more trustworthy systems for the users. Since our main quality attribute is reliability (2.3), we sketch its background and discuss current modeling approaches. At the end of the chapter, we outline the available mathematical formalism (2.4) that allows the failure behavior of a system to be expressed and provides a quantitative prediction of its reliability from a software architecture.

2.1 What is Software Architecture?

An architecture is defined in the *Oxford English Dictionary* as a "complex or carefully designed structure of something". Instinctively when 'architecture' is mentioned, most of us think of civil engineering, and in fact, both share common characteristics [Perry & Wolf, 1992]. In the first place, they are the design part of something bigger. This design encompasses the reasoning and decisions made to achieve the requirements outlined by the clients. Secondly, they share the same construction or development stages: requirements elicitation, designing the system and finally constructing a building or developing a software in accordance with the provided design. Thirdly, they also share additional features such as the use of styles as solutions to commonly occurring problems, multiple views to emphasize certain aspects of the architecture and the achievement of non-functional goals (such as reliability, safety, and usability).

However, the analogy between civil and software engineering contains some conflicting points [Taylor *et al.*, 2009]. For example, buildings are tangible artifacts, as one can discern the features of a building just by looking at it, unlike a software system. Software is much more abstract which makes it more difficult to measure and analyze, and as such, hard to evaluate and assess the stakeholder requirements. Moreover, buildings are less malleable than software. We can mold software in ways that are only imaginable in buildings. Thus, the analogy between civil and software architecture inhibits the idea of dynamism and change.

As a means of overcoming the limitations of dealing with dynamism, some authors have proposed an analogy with the wing of a bird [Garlan *et al.*, 2010]. The wing includes a wide range of dynamic properties which reflect constant change, such as its retraction, flap and extension. It consists of a number of physical features, such as feathers, nerves, bones, blood vessels or muscles. However, the wing on its own is much more than the sum of its parts, in its beauty, performance, reliability and lightweight, which as an ultimate goal provides a bird with agility and speed. In this case, the analogy relates to the partitioning of a bigger problem into smaller ones that can be solved independently and work together to achieve the same goal.

Software architecture is a sub-discipline of software engineering since it is concerned with how software systems are designed and built [ISO/IEC/IEEE, 2011]. In the software development process, architecture can be regarded as part of the specification phase since it documents what the system should do and defines its development constraints [Sommerville, 2000]. Moreover, software architecture goes beyond algorithms and data structures, since it encompasses the design and specification of the overall system structure, by comprising the relationships between software elements and their externally visible properties [Bass *et al.*, 1998; Garlan & Shaw, 1994].

2.1.1 Architectural Constituents

Software architecture is responsible for capturing design decisions by specifying system elements, establishing communication between those elements and also creating the synergies with the environment. Moreover, an architecture describes the rationale of the system by encompassing interaction rules, relevant algorithms or even codes.

The basic elements to describe an architecture consist of components, connectors and configurations. In particular, a component represents a unit of computation which can be a single operation, such as a function, a class or a set of classes that share the same interfaces or functionality, or even a complex operation such as an entire system. Each component requires the specification of its interfaces (which are usually called ports) to describe the type of services that this component depends on and provides to other system components. Connectors are responsible for the interactions between components and distributing data among attached processing units. As a requirement, each connector must include roles that specify how data is exchanged such as ordering properties or communication format. like communication format or orders of interaction. An architectural configuration describes the topology of the system by specifying associations between components and connectors, usually called attachments, and their constraints as restrictions on how components and connectors are bound.

Components, connectors and the architectural configuration can be annotated with properties that hold textual information. These annotations aim to further document each architectural element with relevant information, such as design decisions to be transmitted to other development stages, or even to extend them by specifying non-functional properties, such as the response time or the throughput of a particular processing unit.

2.1.2 Architectural Styles

In software engineering, both practitioner and researcher communities use widely known and reusable architectural solutions to solve recurring problems. In the same way, software architecture communities use architectural styles as a set of design decisions that solve a particular problem in specific development contexts and systems. To be more precise, a style encapsulates design decisions from systems that display a common behavior. As an example, filtering a log file in Unix through the command "*cat* file.log | *grep* word" shows common characteristics of a pipe-and-filter style. In the following sections, we introduce a subset of a wide range of styles. For a more complete list and description of architectural styles, the interested reader should consult [Taylor *et al.*, 2009] or [Garlan & Shaw, 1994].

2.1.2.1 Pipe-and-Filter

This style is most applicable in applications that require a defined series of independent computations to be performed on data. A component, *filter*, reads a data stream as input, applies a set of transformations and produces a data stream as output. Each filter computes data incrementally and normally starts outputting the stream before all the input has been consumed. A real world example of a pipe-filter architecture is an expression written in the Unix shell using the pipe symbol '|'.

Figure 2.1 illustrates a pipe-and-filter style that reads a log file, parses lines with the word "error" and stores them in another log file. In this example, each component represents a filter which reads data streams in its inputs, parses them and produces streams of data in its output. Data is transmitted among components by pipes which are represented through connectors.

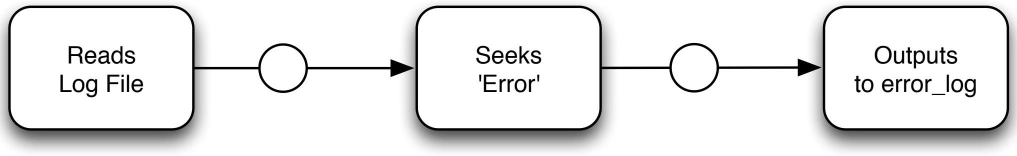


Figure 2.1: Pipe-and-filter style

2.1.2.2 Batch-sequential

Batch-sequential design dates back to the old programming paradigm where different programs communicated through magnetic tapes. As such, each functional step is a separate program which terminates before handling the data for another program. Although it is an old style, it is currently in use, especially when a program has to invoke external scripts or programs to complete its tasks.

To illustrate this style, Figure 2.2 shows the process of updating client accounts through daily transactions. A magnetic tape holds the daily transactions which are sorted by account numbers and then the transactions are processed through the tape that holds every bank account.

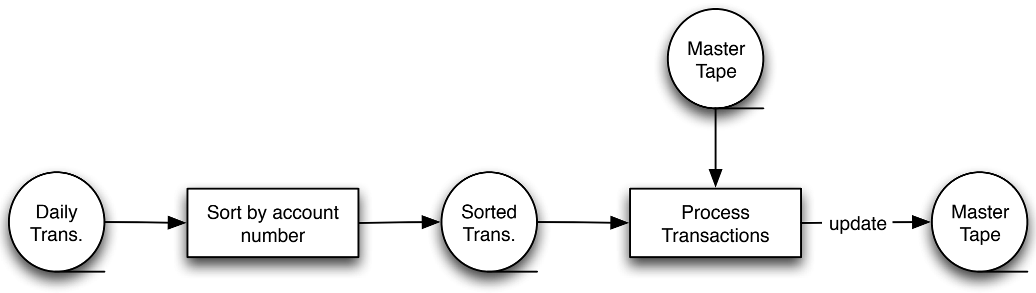


Figure 2.2: Example of a Batch-sequential style

Pipe-and-filter and batch-sequential styles may look similar, but they differ in so far as the former components can execute simultaneously and the latter have to terminate before passing on the data.

2.1.2.3 Call-and-Return

Call-and-return architectural styles are characterized by passing the control to other elements which process data and returning the control to the caller component. This style originated sub-styles like main-program-and-subroutine, remote procedure calls and even object oriented programming. The former is characterized by decomposing a program into smaller modules, as illustrated in Figure 2.3 where the main program invokes its sub-modules. Remote procedure calls behave in a similar way to main-program-and-subroutine systems, but instead of invoking internal modules, contact computers connected via a network. The latter, object-oriented systems are a modern version of call-and-return and include a bundle of services for other components, generally in the form of an interface.

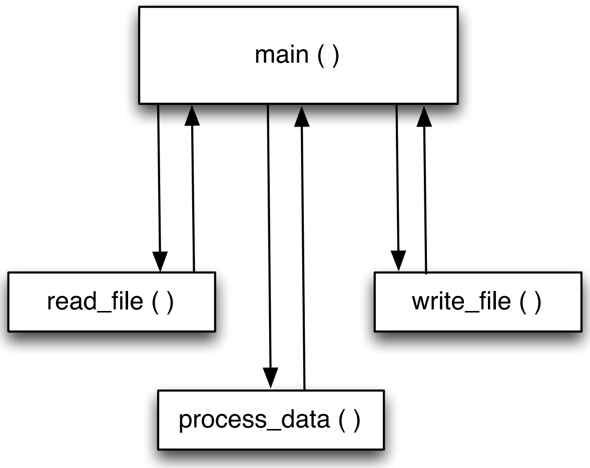


Figure 2.3: Example of a Main-Program-and-Subroutine systems

2.1.2.4 Parallel

In a parallel style, a computational task is decomposed into smaller sub-tasks that can be processed independently. These independent tasks can be computed simultaneously by different processes, threads or even computers to improve performance. At the end, the results are combined and form the outcome of the original large task. The goal of this style is to obtain a higher performance and process large amounts of data in a short period of time. Figure 2.4 illustrates the parallel style by decomposing a task into sub-tasks which can be computed by different sources simultaneously.

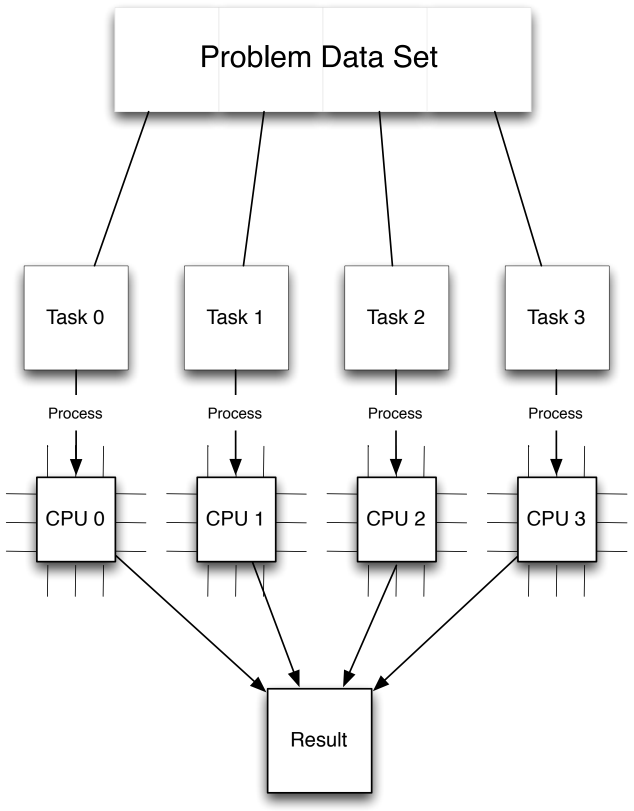


Figure 2.4: Example of a Parallel style

2.1.2.5 Fault-Tolerant

One typical measure to implement fault-tolerance involves the replication of computational resources. If a component fails, the redundant one executes the assigned task and the system can continue to operate properly. Figure 2.5 illustrates an example of an everyday fault-tolerant mechanism. The domain names of the Internet use fault-tolerance mechanisms by requiring the specification of two Domain Name System (DNS) servers. Each server holds a copy of a DNS zone consisting of all the registered domains and sub-domains for that particular DNS server. The primary server holds a master copy of the zone data and the secondary server receives periodical updates, and acts as a redundant server. Whenever the primary server fails, the secondary takes over its functions and thus allows the system to be reachable by visitors.

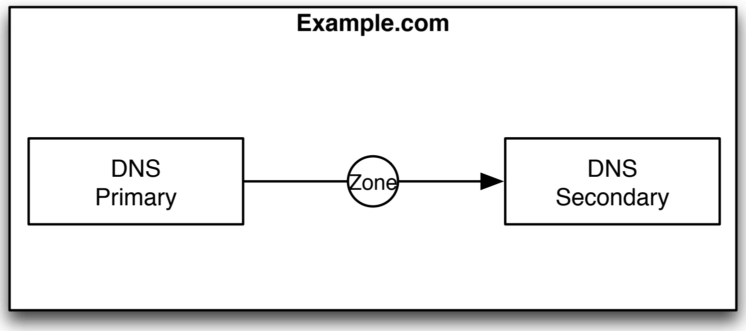


Figure 2.5: Example of a Fault-tolerant style

Due to the technological advances and the adoption of new methods and paradigms, new architectural styles are constantly being designed to deal with modern problems. The architectural styles outlined in this section are a subset of a long list of classical styles which are still commonly used in today's programming techniques. The next section explains how architectures can be formally described by taking account of their elements and architectural styles.

2.1.3 Architecture Description Languages

Architecture Description Languages (ADLs) allows us to formally describe and represent a system architecture by rigorously specifying its structure and behavior through the use of specific notations [Issarny & Zarras, 2003]. The resulting document of an ADL also serves as a means of communication between the people involved in the project, such as designers, developers and stakeholders and even as an important document for the maintenance team.

According to the ISO/IEC/IEEE 42010 Standard in 2011 entitled "Systems and Software Engineering – Architecture Description" an ADL shall specify:

- Concerns about the system when applicable, such its purposes, development and deployment feasibility, potential risks or even the maintainability;
- Involve stakeholders in the previously specified concerns together with developers, owners, users, maintainers;
- Any architectural viewpoint which includes information on the architecture techniques that are used to create, interpret or analyze the system.
- The correspondence rules through which architectural elements must comply, otherwise the architecture may not conform to the requirements.

In short, an ADL should provide explicit support for modelling components, connectors, interfaces and configurations (in the form of properties or rules to validate the system). Taylor *et al.* [2009] distinguish the current existing ADL tools between those that are domain-specific and those that are extensible. The former tools are used in specific contexts where ADLs are optimized to describe architectures belonging to a particular domain or style. For example, RAPIDE [Luckham *et al.*, 1995] is an ADL specifically designed to develop or specify systems that communicate through events. Architecture Analysis and Design Language (AADL) [Feiler *et al.*, 2006] is another domain-specific ADL that is more suitable for modeling embedded and real-time systems, such as automotive and medical systems. The latter type, extensible ADLs, provides a basic set of constructs (*e.g.*, components, connectors, interfaces, rules) to specify an architecture. In addition, these constructs can be extended to support new styles and user-defined constructs that can be applied to general-purpose systems. Acme [Garlan *et al.*, 1997] is an example of an extensible ADLs that makes it possible to specify and extend a set of basic constructs that can express the design decisions of a system. In addition, Acme provides the Acme Tool Developer's Library (AcmeLib) [Garlan *et al.*, 1997] which allows third-party applications to read or modify an architectural model. In short, this library gives support to analyze and validate a system described in Acme.

In this work we decided to use Acme since it is a general purpose ADL [Taylor *et al.*, 2009] that can be applied to more contexts and because it supports validation and assessment techniques through AcmeLib. Conducting an analysis and validation in an architecture is important to assure its quality attributes; these are outlined in detail in the following section.

2.1.4 Software Quality Attributes

One of the primary concerns when developing a software system is to meet functional requirements. In other words, developers must ensure that the final product conforms to the required functions and behavior defined by the stakeholder. However, while the functional properties are essential in the software development process, they are not sufficient. Software developers must also provide non-functional properties[1] as well. For example, a smartphone is a daily-use device that allows the user to carry out several functions to meet the functional requirements, like listening to music, browsing the web or making phone calls. However, from a non-functional perspective, if the interface performs poorly, the user may become dissatisfied and lose faith in the equipment brand or operative system, and as result cease to use the device or decide to buy another brand in the future. In this case, this type of devices has a quality attribute related to performance that is vital for the daily use of these systems.

Software architecture provides the foundations for the achievement of non-functional properties. More specifically, the architectures provide designers with means to codify non-functional properties, employ good design practices, assure compliance with the requirements and maintain their traceability throughout the software lifespan.

Quality attributes should be considered carefully by the stakeholders and the architect, since none of them can be achieved in isolation and they have to be combined. Additionally, the achievement of one quality attribute may have a positive or negative effect on another. One example of this trade-off between quality attributes is the achievement of portability, which usually has a performance drawback. Described in more detail, one of the techniques employed to achieve portability involves isolating the system dependencies, which introduces overhead into the execution of the system, and thus reduces its overall performance.

[1]The terms 'quality' and 'non-functional property' mean the same and are used interchangeably throughout this document

In short, the non-functional property of a system imposes a constraint on how the system delivers its services. This property can be measured quantitatively or qualitatively and it tends to be hard to measure precisely. There is no strict number of non-functional properties, but they include reliability, security, efficiency, availability and many other *–ilities.* One of the most important non-functional properties is dependability [Taylor *et al.*, 2009], which is discussed in greater depth in the next section.

2.2 Dependability

Dependability is one of the quality attributes that cause major concern when designing a system. [Taylor *et al.*, 2009]. Dependability is closely related to trust since it is a measure that determines how a system or user can justifiably place trust in the services delivered by another system. For example, the dependence of system A on B leads one to assume that the dependability of A is directly influenced by the dependability of B [Avizienis *et al.*, 2004; Lyu, 1996].

Dependability is a broad concept that can be split into three elements: attributes, threats and means. *Attributes* is defined as a set of non-functional properties to achieve, *threats* is described as the circumstances that have an adverse effect on dependability, and *means* is characterized by the techniques used to increase the trustworthiness of a system.

2.2.1 Attributes

Dependability is a broad concept that includes the following non-functional properties, defined by Avizienis *et al.* [2004]:

- Availability – readiness to provide a correct service;
- Reliability – continuity of correct service;
- Safety – absence of catastrophic consequences on the user or the environment;
- Integrity – absence of improper alterations to the system;

- Maintainability – ability to undergo modifications and repairs.

The dependability requirements may vary depending on the designed system and its environment. For instance, availability is a property that is transversal to all systems, but an online server hosting a critical application may require a greater degree of availability than other servers hosting a website. Thus, when designing a system an important phase is to elicit its specific requirements and ensure that quality trade-offs are minimal.

2.2.2 Threats

The research community regards threats to dependability as a categorization of issues that can be faced by every computer system. Moreover, these issues may have serious effects on the system or the user, but ultimately will lead to a decline in the system's trustworthiness. The classification of threats to dependability is listed below:

- Fault – this is a defect in the system and informally also referred to as a *bug*. A fault can be in either one of two states: dormant or active. A fault is dormant if it is present in the system and only activated by an invocation.

- Error – this occurs at runtime when the system enters in an unexpected state due to the activation of a fault. When an error has not been identified as such, it is considered to be *latent*. Otherwise, it enters the *detected* state. Moreover, an error may disappear before it has been detected and, in a latent state, the error may be propagated to other components.

- Failure – this takes place when the error is noticeable externally. It is caused when a service is delivered that deviates from the correct one or an abnormal event occurs (*e.g.*, crash, hang).

In practical terms, a dormant fault is a defect that is codified in the software as a faulty instruction or data. Whenever the component in which the fault is located is invoked and the faulty instruction is triggered through a code sequence, data or an input pattern, the fault becomes active and produces an error. In its turn, an error only leads to a failure if it crosses the boundary of the system, and becomes visible to the outside. A failure is thus, a transition from the correct to an incorrect service.

2.2.3 Means

A way to provide more dependable systems is to avoid service failures that are more frequent or more serious than is acceptable. Following, we present a set of means to attain a more dependable system:

- Fault prevention – preventing the occurrence or introduction of faults;
- Fault removal – reducing the number or severity of faults;
- Fault tolerance – avoiding service failures in the presence of faults;
- Fault forecast – estimating the present number, future occurrence and implications of faults.

The previously specified quality attributes (*i.e.*, availability, reliability, safety, integrity and maintainability) should be attained to achieve a more dependable system. With this in mind, the following section details one of these quality attributes – reliability – by describing its concept and how it can be modeled in an early phase.

2.3 Reliability Theory

Reliability was established as a field of study during the Second World War much because of the problems experienced with military equipment that failed very quickly. Many of the problems were related to electronic equipment that had an operational lifetime of only a few hours. As a result, much of the military equipment failed even before it went into service. To circumvent this problem, the United States (US) government set up a number of research groups to improve the reliability of electronic equipment. In 1950 the US Department of Defense carried out a review of all the electronic equipment used by the army, navy and air force. Afterwards, they brought together a group of representatives from the electronics industry to form the Advisory Group on Reliability of Electronic Equipment (AGREE). The final goal of this joint group was to produce a set of documents establishing good working practices and ensuring the reliability of electronic equipment. Today, the reliability of electronic equipment has increased enormously as even be witnessed in household appliances. However, although electronic reliability has been studied since the 1950s, software one is still in its infancy and is becoming a major concern since it is being applied to critical systems such as automobiles, planes or spacecrafts [Storey, 1996].

In the next section we examine in detail the concept of reliability and how it can be calculated for individual elements as well for the whole system.

2.3.1 Concept

Reliability can be described as the continuous and correct delivery of a service or an assurance of the probability of failure-free operations [Storey, 1996]. In addition, it can be expressed through two different types of behavior: continuous and discrete-time [Lyu, 1996]. The former involves the notion of time and its distribution is expressed through the following formula where $R(t)$ represents the reliability over time, t denotes the time elapsed since the start of the execution and λ expresses a constant failure rate.

$$R\left(t\right) = e^{-\lambda t}$$

The latter, discrete-time, leverages the notion of time by using the failure probability to estimate reliability. It assumes that a failure may occur during the process of an input or in the control transfer between two modules. Moreover, reliability (denoted by R) expresses the probability of delivering a correct service and is quantified as the number of correct delivered responses over the total number of requests.

$$R = Pr\left\{\text{no system failure at output} \mid \text{no failure at input}\right\}$$

The formulas mentioned above express the reliability of individual components. However, a system is usually composed of more than one component and includes the notion of their interdependence. With this in mind, the next section explores methods to calculate the reliability of a whole system by examining a set of components and their connections.

2.3.2 Modeling

Usually dependable software components will, on average, result in more dependable systems. However, this should not be regarded as a law, since highly dependable systems may comprise unreliable components and dependable components do not always result in dependable systems [Taylor *et al.*, 2009]. This difference illustrates the need to quantitatively predict reliability in an early phase of the software development, but also requires an analysis and suggestions on where improvements can be made in the architecture.

With this in mind, we examined a set of methods that allow reliability to be modeled from a software architecture and they all involve stochastic modeling approaches. Stochastic systems encompass the occurrence of random events, aiming to calculate or analyze processes that cannot be determined precisely due to the unpredictability involved. These systems are usually applied in contexts that involve a source of randomness, such as natural events, economics, science or engineering.

With regard to software and stochastic models, the European Space Agency (ESA) in 1986 conducted a series of reliability studies and concluded that software failures appear to be "random" to the observer. Moreover, they also state that "the term 'reliability' is meaningful when applied to a system that includes software and the process can be modeled as stochastic"[European Space Agency (ESA), 1988].

With this in mind, several studies applied stochastic models to estimate reliability on the basis of a software architectural description [Brosch *et al.*, 2011; Cortellessa, 2002; Reussner & Heinz W., 2003; Yacoub *et al.*, 2000]. Among the first to propose architectural reliability modeling by using Markov chains, was Cheung [Cheung, 1980] and several surveys have been conducted since then [Gokhale, 2007; Goševa-Popstojanova & Trivedi, 2001; Immonen & Niemelä, 2008; Pengoria *et al.*, 2009].

According to these surveys and as depicted in Figure 2.6, the reliability assessment of software architectures can be performed through two different approaches which combine the architecture with the failure behavior [Goševa-Popstojanova & Trivedi, 2001]: path and state-based models.

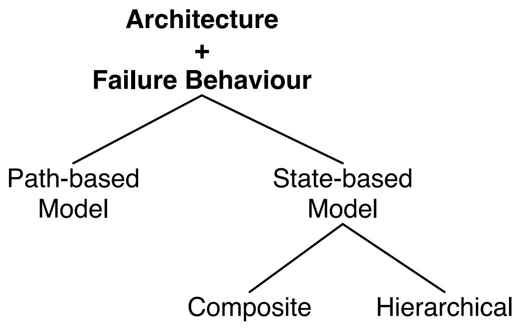


Figure 2.6: Approaches to combine the architecture with the failure behaviour

2.3.2.1 Path-Based Model

This class of models estimates reliability by combining software architecture with the failure behavior. In other words, reliability is computed by traversing the possible execution paths of the program. The overall reliability is then calculated by averaging all the computed reliability paths. Architecturally speaking, this model assumes a knowledge of the different system paths and their frequencies. Although this information may not be accessible at every development stage, it can be obtained experimentally, by testing or algorithmically Goševa-Popstojanova & Trivedi [2001].

For example, Shooman [Shooman, 1976] proposes a method to calculate the reliability of modular programs that encompass the execution frequency for each path. In detail, this model assumes the knowledge of all paths in the system, denoted by m, and the execution frequencies f_i of each path i. In addition, the failure behavior is denoted by the failure probability q_i of each execution path

i. Then, the total number of failures n_f in N test runs is obtained through the following formula:

$$n_f = \sum_{i=1}^{m} N \cdot f_i \cdot q_i$$

The reliability of the system (q_0) is given by:

$$q_0 = \lim_{N \to \infty} \frac{n_f}{N} = \sum_{i=1}^{m} f_i \cdot q_i$$

This model has some drawbacks which makes its applicability of limited value. In detail, when a reliability problem arises, this method allows us to identify a path of the architecture that requires urgent attention and improvement. However, identifying specific components within that particular path can be a hard task. In addition, the path-based model cannot be applied to systems that contain loops in the architecture, since it will only provide an approximate estimation of the system reliability [Goševa-Popstojanova & Trivedi, 2001].

2.3.2.2 State-Based Models

State-based models assume that transitions comprehend a Markov property, meaning that at any time the future behavior of components or transitions between them is conditionally independent of the past behavior. These models require a knowledge of the following features:

- Software architecture – this describes how the components are arranged in the system and how they interact with each other. In addition, the flow of information, like input or output data must be also specified.
- Usage (or operation) profile – this explains how the system is used and may include the transition probabilities between modules or the execution time for each module.

- Failure behavior – this provides a quantitative measure of the number of failures for each component in the system. This number can be defined in one of two ways: component reliabilities and failure rates (constant or time-dependent). The first is defined by a probability expressing the correct service delivery, while the second expresses a time frequency in which a component fails.
- Interfaces – this specifies the interaction between components. Usually their failure behavior is assumed to be perfect, but it can also be defined as the failure behavior of the components.

Moreover, state-based models can be divided into composite [Cheung, 1980; Reussner & Heinz W., 2003] and hierarchical [Gokhale & Trivedi, 2002] methods which are outlined below:

Composite Method. The architecture of the system and the failure behavior of its components are combined in a single model. In other words, the components and connectors may be represented as states in the model. With regard to the system failure behavior, it can be encoded by assigning probabilities between state transitions. A failure may be represented as the transition from a correct state to a faulty one.

Hierarchical Method. This method assumes that the architecture and the failure behavior are detached; more specifically the architecture is solved first and then the failure behavior is superimposed as a function to determine the overall reliability. For example, an architecture can be modeled by a semi-Markov process and the failure behavior can be modeled according to a Poisson process [Littlewood, 1979] or by a time-dependent failure rate [Laprie & Kanoun, 1996].

The above methods (Composite and Hierarchical) are part of the state-based models approach and both present accurate results when compared to the actual reliability of a system [Goseva-Popstojanova *et al.*, 2005]. With this in mind, one may select a method over another, since they present close results. In this case, we decided to use the composite method over the hierarchical, since it combines the architecture with the failure behavior which facilitates the translation from the ADL to the state-space based model.

The modeling approaches presented above allow reliability to be assessed quantitatively from a software architecture. However, their application requires a mathematical formalism to solve these models and obtain a reliability figure. For this reason, the next section explores the mathematical formalisms and explains how reliability can be determined.

2.4 Probabilistic Model Checking

A promising approach to detect and correct possible errors in the early stages of software development is to model programs and systems by specifying their behavior and checking their correctness, which is often referred to as the 'formal methods' [Emerson, 2008].

According to Emerson [Emerson, 2008] the development of model checking was motivated by the predominant manual activities to verify and prove the theoretical reasoning, using axioms and inference rules. These manual activities involve issues similar to any other task performed by a human: error proneness and time-consuming activities. In view of this, Clarke and Emerson [Clarke & Emerson, 1981] described model checking as:

> "A method to establish that a given program meets a given specification, where:
>
> - The program defines a finite state graph M.
> - M is searched for elaborate patterns to determine if the specification f holds.
> - Pattern specification is flexible.
> - The method is efficient in the sizes of M and, ideally, f.
> - The method is algorithmic.
> - The method is practical."

Model checking employs formal verification techniques to guarantee whether a particular property holds in the modeled software program. For example, model checking can be used to verify if a software enters in a deadlock condition. If a property can be quantitatively assessed, we use probabilistic model checking to verify if that particular property holds. Probabilistic model checking involves the same steps as the model checker: building the model, formal specification of a desired property and checking the model for the satisfaction of that property.

The following subsections introduce different probabilistic models, temporal logics used in these models and the currently used probabilistic model checking tools, as well as, making comparisons based on a different number of properties.

2.4.1 Discrete-Time Markov Chain (DTMC)

The notion of what is currently called a Markov Chain was conceived by the Russian mathematician Andrei Markov, who at the beginning of the 20^{th} Century was investigating the alternation of vowels and consonants in the Eugene Onegin the famous poem by the Russian author Alexander Pushkin. Markov devised a probability model in which the outcomes of each trial only depend on its immediate predecessor. This model turned out to be an excellent description of the problem which the mathematician was trying to solve, that give an accurate estimation of their frequency of appearing [Tijms, 2003].

Discrete-Time Markov Chains (DTMCs) only consider state transitions that occur at fixed time intervals and can be defined as follows [Kwiatkowska *et al.*, 2007]:

Definition 1 (DTMC) *A discrete-time Markov Chain is a tuple $M = (S, \overline{s}, P, L)$, where:*

- *S is a finite, non-empty set of states;*
- *$\overline{s} \in S$ is the initial state;*
- *$P \colon S \times S \to [0, 1]$ is the transition probability matrix where $\Sigma_{s' \in S}\mathbf{P}(s, s') = 1$ for all $s \in S$;*

- *L: $S \rightarrow 2^{AP}$ is a labeling function which assigns to each state $s \in S$ the set $L(s)$ of atomic propositions that are valid in the state.*

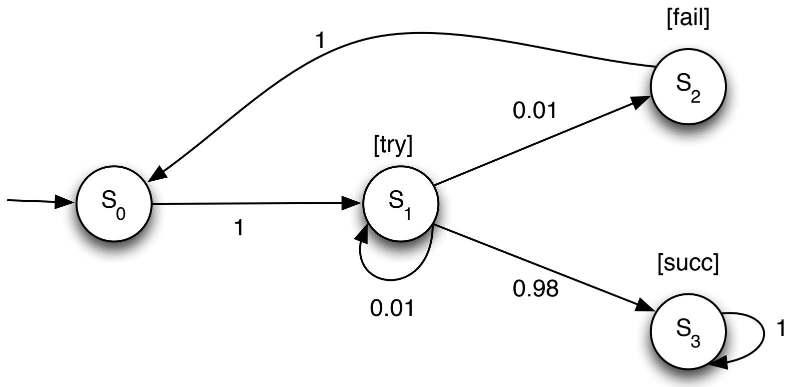


Figure 2.7: Discrete-Time Markov Chain (DTMC)

Figure 7.1 adapted from [Kwiatkowska *et al.*, 2007], illustrates an example of a DTMC $D_1 = (S_1, \overline{s_1}, P_1, L_1)$, where states are represented by circles and their transitions are drawn as arrows, labeled with their associated probabilities. The initial state is $\overline{s} = s_0$, the DTMC has four states, where $S_1 = \{s_0, s_1, s_2, s_3\}$ and the associated transitional probability matrix P_1 is given by Formula 2.1.

$$\mathbf{P}_1 = \begin{pmatrix} & s_0 & s_1 & s_2 & s_3 \\ s_0 & 0 & 1 & 0 & 0 \\ s_1 & 0 & 0.01 & 0.01 & 0.98 \\ s_2 & 1 & 0 & 0 & 0 \\ s_3 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.1}$$

2.4.2 Continuous-Time Markov Chain (CTMC)

Continuous-Time Markov Chains (CTMCs) are very similar to DTMCs differing only in the frequency of the state transitions. In CTMCs a transition can occur at any time, while in DTMCs they correspond to a discrete time-step [Kwiatkowska *et al.*, 2007]. Thus, a CTMC is defined as follows:

Definition 2 (CTMC) *A continuous-time Markov Chain is a tuple $C = (S, \overline{s}, R, L)$, where:*

- *S is a finite, non-empty set of states;*
- *$\overline{s} \in S$ is the initial state;*
- *$R : S \times S \to \mathbb{R}_{\geq 0}$ is the transition rate matrix;*
- *$L : S \to 2^{AP}$ is a labeling function which assigns to each state $s \in S$ the set $L(s)$ of atomic propositions that are valid in the state.*

The transition rate matrix R contains the rates of each pair of states in the CTMC and the transition can only occur if the rate between states s and s' is greater than 0 (*i.e.*, $R(s, s') > 0$). The probability of this transition being triggered within t time-units equals to $1 - e^{-R(s,s')\cdot t}$.

The time spent on each state s, before a transition occurs is defined by $E(s)$, known as the 'exit rate of state s':

$$E(s) = \sum_{s' \in S} R(s, s') \tag{2.2}$$

It is also possible to determine what is the probability of each state s' being the next state to which a transition is made from s, regardless of the time at which this occurs. This is defined by $\mathrm{P}^{emb(C)}$ in Formula 2.3.

$$\mathrm{P}^{emb(C)}(s, s') = \begin{cases} \frac{\mathbf{R}(s,s')}{E(s)} & \textit{if } E(s) \neq \mathit{0} \\ 1 & \textit{if } E(s) = \mathit{0} \\ 0 & \textit{otherwise} \end{cases} \tag{2.3}$$

To perform the analysis of a CTMC $C_1 = (S_1, \overline{s_1}, R_1, L_1)$ an infinitesimal generator matrix is needed which is given by $Q : S \times S \to \mathbb{R}$ in Formula 2.4.

$$Q(s, s') = \begin{cases} R(s, s') & if\ s \neq s' \\ -\Sigma_{s'' \neq s} R(s, s'') & otherwise \end{cases} \tag{2.4}$$

Figure 2.8 illustrates a CTMC example adapted from [Kwiatkowska *et al.*, 2007], which represents a queue of jobs. Each state s_i is a job and there are i jobs in the queue. Initially the queue is empty ($\overline{s} = s_0$), each job arrives with a rate of

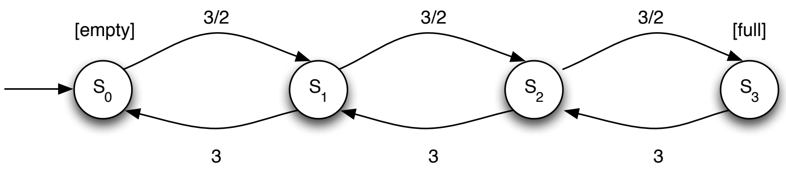


Figure 2.8: Continuous-Time Markov Chain (CTMC)

$\frac{3}{2}$ and are removed from the queue with a rate of 3. The associated transition rate matrix R_1, transition probability $\mathrm{P}_1^{emb(C_1)}$ and the infinitesimal generator matrix are described below.

$$R_1 = \left(\begin{array}{ccccc} & s_0 & s_1 & s_2 & s_3 \\ s_0 & 0 & \frac{3}{2} & 0 & 0 \\ s_1 & 3 & 0 & \frac{3}{2} & 0 \\ s_2 & 0 & 3 & 0 & \frac{3}{2} \\ s_3 & 0 & 0 & 3 & 0 \end{array}\right) \tag{2.5}$$

$$\mathrm{P}_1^{emb(C_1)} = \left(\begin{array}{ccccc} & s_0 & s_1 & s_2 & s_3 \\ s_0 & 0 & \frac{3/2}{3/2} & 0 & 0 \\ s_1 & \frac{3}{3+\frac{3}{2}} & 0 & \frac{\frac{3}{2}}{3+\frac{3}{2}} & 0 \\ s_2 & 0 & \frac{3}{3+\frac{3}{2}} & 0 & \frac{\frac{3}{2}}{3+\frac{3}{2}} \\ s_3 & 0 & 0 & \frac{3}{3} & 0 \end{array}\right) = \left(\begin{array}{ccccc} & s_0 & s_1 & s_2 & s_3 \\ s_0 & 0 & 1 & 0 & 0 \\ s_1 & \frac{2}{3} & 0 & \frac{1}{3} & 0 \\ s_2 & 0 & \frac{2}{3} & 0 & \frac{1}{3} \\ s_3 & 0 & 0 & 1 & 0 \end{array}\right) \tag{2.6}$$

$$Q_1 = \left(\begin{array}{ccccc} & s_0 & s_1 & s_2 & s_3 \\ s_0 & -\frac{3}{2} & \frac{3}{2} & 0 & 0 \\ s_1 & 3 & -\frac{9}{2} & \frac{3}{2} & 0 \\ s_2 & 0 & 3 & -\frac{9}{2} & \frac{3}{2} \\ s_3 & 0 & 0 & 3 & -3 \end{array}\right) \tag{2.7}$$

2.4.3 Absorbing Markov chains

An absorbing Markov chain is a special type of Markov chains where every state can reach one of the absorbing states.

Definition 3 (Absorbing Markov chain) *A state s_i is called absorbing if it has a self-loop transition with rate of* 1.0 *meaning that after entering, it is impossible to leave. Other states than the absorbing ones are called transient. A Markov chain is absorbing if it has at least one absorbing state and if from every state it is possible to reach at least one absorbing state in a finite number of steps.*

An absorbing Markov chain allows us to determine (a) the probability that the process will end in a given absorbing state; (b) the number of steps, on average, that are required for the process to be absorbed; and (c) the number of times, on average, the process will be in a particular transient state.

The transition matrix of an absorbing Markov chain with r absorbing states and t transient states has the following canonical form.

$$P = \begin{bmatrix} Q & R \\ 0 & I \end{bmatrix}$$

In the transition matrix P, I is an r-by-r identity matrix, 0 is an r-by-t zero matrix, R is a nonzero t-by-r matrix and Q is an t-by-t matrix. P accounts for the fact that the first t states are transient and the last r states are absorbing.

2.4.4 Markov Decision Process (MDP)

Markov Decision Processs (MDPs) extend Markov chains by adding non-determinism, which involves randomness in future state transitions and does not produce the same output from a given starting-point or initial state. Hence, MDPs permits both probabilistic and non-determinism choices. Probabilistic choices serve to model and quantify the possible outcomes of randomized actions such as throwing a dice, tossing a coin or sending a message through a lossy communication channel. In addition, an example of the usage of probabilistic choices is, for instance, a coffee vending machine which sells normal and decaffeinated coffees, assigning probability $\frac{8}{10}$ for the normal choice and $\frac{2}{10}$ for the decaffeinated choice. This example requires statistical experiments on the behavior of the environment to obtain appropriate distributions that model the choices of sold coffee. However, if this information is not available or is required to guarantee the system properties, the choice will be made by a non-deterministic model and, therefore, use MDPs instead of Markov chains [Baier & Katoen, 2008; De Alfaro, 1998].

According to Baier & Katoen [Baier & Katoen, 2008], MDPs can be defined by the following definition:

Definition 4 (MDPs) *A MDP is a tuple $M = (S, Act, P, s_{init}, AP, L)$, where:*

- *S is a countable set of states;*
- *Act is a set of Actions;*
- *$P : S \times Act \times S \rightarrow [0, 1]$ is the transition probability function such that for all states $s \in S$ and actions $\alpha \in Act$:*

$$\sum_{s' \in S} P(s, Act, s') \in \{0, 1\}$$

- *$s_{init} : S \rightarrow [0, 1]$ is the initial distribution such that $\sum_{s \in S} s_{init}(s) = 1$;*
- *AP is a set of atomic propositions and $L : S \rightarrow 2^{AP}$ a labeling function.*

An example of a Markov Decision Process (MDP) is depicted in Figure 2.9, adapted from [Baier & Katoen, 2008].

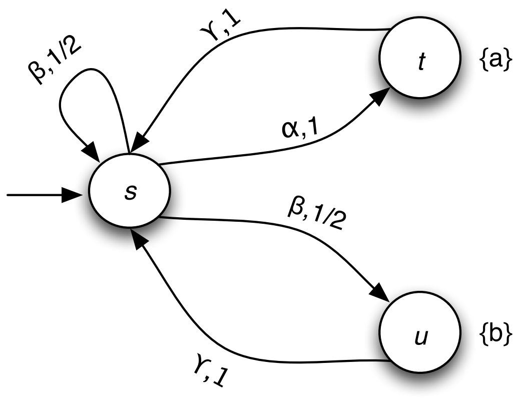


Figure 2.9: Markov Decision Process (MDP)

It can be seen in the above example that State s is the initial state, so $s_{init}(s) = 1$. The initial state, s, can choose between two different actions $\{\beta, \alpha\}$ and the other states can only choose one action $\{\gamma\}$. More precisely, below are specified the sets of enabled actions for each of the states found in the example.

- $Act(s) = \{\alpha, \beta\}$ with $P(s, \alpha, t) = 1, P(s, \beta, u) = P(s, \beta, s) = \frac{1}{2}$
- $Act(t) = Act(u) = \{\gamma\}$ with $P(t, \gamma, s) = P(u, \gamma, s) = 1$

2.4.5 Temporal Logic

According to Emerson [Emerson, 2008], a model checker comprises several steps such as building the model, providing a formal specification of the desired properties that reveal the behavior of the system and checking the model for the validation of those properties. The building stage of the model was addressed in the previous subsections and in this subsection we focus on solving the issue of how to provide formal specifications of the intended properties. In particular, temporal logic is used to describe the behavior of the system through rules and symbolic elements representing propositions.

Below we define different temporal logics, which can be applied to the probabilistic model checkers described in the previous subsections. However, the specification of their syntax and semantics specification go beyond the scope of this thesis, the interested reader can consult the following literature [Baier & Katoen, 2008; Emerson, 2008; Tijms, 2003].

Probabilistic Computation Tree Logic (PCTL) is a probabilistic extension of the Computation Tree Logic (CTL) and is applied for DTMCs. This logic is useful to state soft deadline properties such as “what is the maximum probability of reaching an absorb correct state?”.

Linear Temporal Logic (LTL) is a modal temporal logic where each point in time has a unique successor from an infinite sequence of states. LTL allows to describe properties about the future of the paths, such as “a condition will be true until another fact becomes true”.

Continuous Stochastic Logic (CSL) is a time-bounded property which includes an operator to reason about steady-state probabilities. This logic is used to express properties over CTMCs which have a rational time bound, for example: "the probability of a system producing an error within 4 time-units is less than 10^{-2} expressed through $\mathbb{P}_{<0.01}(\cup^{\leqslant 4}\text{ error})$"

2.4.6 Tools

Several tools are available to solve the probabilistic models described previously, in this section we examine some contemporary tools and provide a comparison between their properties.

Prism is a free and open-source tool for formal modeling and analysis of systems which display a random or probabilistic behavior [Hinton *et al.*, 2006]. In addition, PRISM uses a high-level state-based language, the PRISM language, to describe models and also supports three types of probabilistic models: DTMCs, CTMCs and MDPs.

Rapture is a tool that performs verification of quantified reachability properties over Markov Decision Processes (MDP). The originality of Rapture when compared with Prism, is that Rapture provides two reduction techniques that limit the state space explosion problem [Jeannet *et al.*, 2002].

Although Rapture looks promising, it lacks documentation and has not been publicly released, since there is no available site or information regarding its properties. As a result, Rapture integrates the list of probabilistic model checker tools, although it has not been possible to determine what are its promising features.

Markov Reward Model Checker (MRMC) is a model-checker developed collaboratively by two European Universities: University of Twente in Netherlands and the RWTH Aachen University in Germany. This tool allows the use of reward models and both discrete- and continuous-time Markov chains. In addition, it also supports reward extensions of Probabilistic Computation Tree Logic (PCTL) and Continuous Stochastic Logic (CSL), by enabling automated checking of properties concerning long-run and instantaneous rewards [Katoen *et al.*, 2009].

Process Analysis Toolkit (PAT) implements several model checking techniques, supports Linear Temporal Logic (LTL) properties, and carried out refinement and probabilistic model checking. Moreover, it also allows the verification of deadlock-freeness, reachability analysis and state or event linear temporal logic checking.

According to [Sun *et al.*, 2008], "PAT is capable of verifying systems with a large number of states and outperforms the popular model checkers in some cases", which makes it very useful and has thus been included in the comparison.

Tool Comparison. Table 2.1 displays the tools previously described, omitting Rapture owing to a lack of information. In addition, there are more tools available regarding model checking, but we decided a) only to examine the ones that use probabilistic model checking, b) to order them in accordance with their software license and c) only give preference to those that have a free or open-source license.

Regarding the tool adopted in our work, we decided on the Prism model checker tool due to its wealth of documentation and the fact that it is open source. This last advantage allowed us to build a customized library which could be employed in our solutions independently of the operating system and with a low performance overhead.

2.5 Summary

This chapter has sketched the background of the questions and issues that underpin this thesis. We began by defining software architecture and revealing its importance to the software development life-cycle. Moreover, we showed how the quality of a system can be expressed in the architecture and how more dependable systems can be obtained from design. The main quality that is being investigated in this thesis – reliability – was also introduced along with the methods for its assessment by means of a software architecture. At the end of this chapter, we

Name	**Prism**	**MRMC**	**PAT**
Modelling language	Prism Language, Plain MC	Plain MC	CSP#, Timed CSP, Probabilistic CSP
Language Properties	CSL, PCTL, PCTL*, LTL	CSL, CSRL, PCTL, PRCTL	LTL, Assertions
Graphical Interface	Yes	No	Yes
Software License	Free and Open Source	GNU Public License	Free
Programming Language	Java	C	C#
Operating System	Windows, Unix, Mac OS X	Unix, Windows, Mac OS X	Windows and other OSes with mono

Table 2.1: Probabilistic Model Checking tools

listed the mathematical formalisms required to calculate reliability along with some tools that simplify the process of probabilistic model checking.

Most of today's guidelines to make software architectures more dependable, are concerned with techniques to design the system to resist or recover from faults by providing reflection capabilities, making use of exception handling and overseeing the external component interdependencies. However, automated methods to assess a software architecture with regard to its dependability and analyze the presence of architectural weak points (*i.e.*, bottlenecks) are still in their infancy and absent from ADL tools. With this in mind, in the next chapter we propose novel techniques to accomplish these tasks – by predicting and analyzing reliability early in the software development life-cycle through an automated approach.

Chapter 3

Automated Reliability Prediction

Along with software development, a need has arisen to assure if the product under implementation meets the required quality goals. Otherwise, the final product may not conform to the requirements which will set back the development process to a previous phase, increase the project costs and delay product delivery. This quality assessment can take place at more than one development stage. For instance, in the design stage architects plan and describe the system in an early phase even before any implementation or testing has been conducted. Thus, assessment tools and methods should allow architects to identify possible issues and assure them that the desired standards are being met. Moreover, during an evolutionary stage, developers can make use of quality assessment methods to determine which components should be upgraded or test if the evolved architecture improves the system quality.

These quality assessment methods have been proposed in recent decades [Cheung, 1980], but they still rely on manual tasks to be built. Currently, very few of the non-functional requirements are automatically checked. This manual checking activity is prone to errors and is time-consuming due to the overwhelming complexity of the designs. Current approaches lack the ability to set out automated methods, translation techniques or apply theoretical methods to the industry or practitioner community.

We seek to bridge the gap between both the research and practitioner communities by adopting an approach that assesses the reliability of a software architecture. More specifically, our approach takes as input an architecture that is described through an Architecture Description Language (ADL) and generates a stochastic model. This model makes it possible to predict the reliability of the system quantitatively, by providing the means for architects to test and validate their designs.

In this chapter we identify and formally specify the architectural elements required by our approach (3.1). Following this, we define reliability (3.2), along with an explanation of the applied mathematical formalism needed to build the stochastic models (3.3). To exemplify the construction of these models, we take an example from a demonstrative software architecture (3.4). The translation from an ADL to the generated stochastic model is performed by a translation process (3.5) which also encompasses different architectural styles that can be applied to a system (3.6). We list the required techniques or tools (3.7) needed to make the proposed method available for architects and allow them to automatically test and predict reliability from the designed architecture. Finally, we outline the assumptions that we rely on and may threat the validity of the proposed approach (3.8), before presenting the related work (3.9).

3.1 Architectural Identification and Specification

Architecture Description Languages (ADLs), presented in Chapter 2.1.3, allow to support annotations to specify key properties for analysis and the validation of quality attributes. ADLs like Acme [Garlan *et al.*, 1997], Wright [Allen & Garlan, 1996] or AADL [Feiler *et al.*, 2006], provide a semantically narrow, formal and unambiguous specification of a system that embodies design decisions with a high level of rigor. In our approach we adopted Acme as the input for our reliability prediction method owing to its general purpose and extensible ADL [Taylor *et al.*, 2009]. The required elements, properties and annotations to accurately predict reliability are specified in the Z notation [Potter *et al.*, 1996; Spivey, 1989]. We chose Z rather than other formal notations like Vienna Development Method (VDM) [Bjørner & Jones, 1978] or Communicating Sequential Processes (CSP) [Hoare, 1978] since its definition is close to set theory and it has been successfully used in software engineering for over two decades [Potter *et al.*, 1996]. Formally annotating a system has more advantages than natural languages by providing a rigorous demonstration through mathematical proof and a more unambiguous and clear meaning than natural discourse. In a practical sense, Z allows future researchers to implement their own reliability prediction techniques and extend our notation to other quality attributes without ambiguity and in a rigorous way.

An architecture comprises what is essential or fundamental to a system in relation to its environment. Its description is a work product from the standpoint of architects and may encompass system constituents (*e.g.*, components, connectors), about how they are organized, their design requirements and principles regarding evolution [ISO/IEC/IEEE, 2011].

An architectural model is a tuple $A = (C,\ Con,\ Att,\ Prop)$, where:

- $C = \{c_i\}$ is a finite set of Components. A component represents a unit of computation which can be a single operation, such as a function, a class or a set of classes that share the same interfaces or functionality, or even a complex operation as an entire system. We refer to a component as a tuple $c_i = (IP, OP, Prop, Rep)$ where:

  $IP = \{ip_j\}$ is a finite set of input ports. Each input port represents the incoming data to be processed by the component;

  $OP = \{op_k\}$ is a finite set of output ports. Each output port represents the data sent from a component after being processed;

  Prop is a set of properties annotating the component with data regarding its behavior. Each property is a tuple that holds information about the name of the property, its type and its value. For example, a component may hold a property representing its response time which is a float and a value representing its current or average response time value.

$$\begin{aligned} Enum &= \{String\} \\ PropType &::= Float \mid Integer \mid String \mid Enum \\ Prop &= \{name, value : String\,;\ type : PropType\ \bullet \\ &\qquad\qquad\qquad (name, type, value)\} \end{aligned}$$

Rep is a representation that specifies the internal behavior of a component c_i. This internal representation is optional in each component and when it exists describes a sub-architecture model that specifies in detail the functionality of that component and it is modeled as a an architecture.

$$Rep = A' \mid \varnothing \Leftrightarrow Rep = (C',\ Con',\ Att',\ Prop') \mid \varnothing$$

- $Con = \{con_i\}$ is a finite set of Connectors. Connectors are the architectural elements responsible for the interactions between components, distributing data among attached components. Each connector is represented by the tuple $con_i = (R, Prop)$:

  $R = \{r_i\}$ is a finite set of Roles. Each Role is responsible for coordinating the communication between the connector and a set of components, by specifying the communication protocol, assurance properties and the rules about interaction ordering or format. It is specified as a tuple $r_i = (Prop)$ where it defines its own properties. The connector role is bound as one-to-one with a component port and each connector must have at least two roles.

  Prop is a set of properties annotating the connector with data regarding its behavior and defined as the same type specified above.

- *Att* is an Attachment showing how components and connectors are bound together. In more detail, an Attachment is a tuple that specifies a component and its port (either input or output) that are connected to a role of a connector. As a result, one can understand how data traverses within the architecture and its elements.

$$Att = \{c_i \in C\, ;\, p \in IP(c_i) \cup OP(c_i)\, ;\, con_l \in Con\, ;\, r_m \in R(con_l) \quad \bullet \quad (\,(c_i,\ p),\ (con_l,\ r_m)\,)\}$$

- *Prop* is a set of properties that annotates the architecture with data regarding requirements or design principles. Each property is defined as the same *Prop* type previously specified.

This formal specification of the architecture in Z enables us to translate it unambiguously to a stochastic model, allowing to automatically predict reliability from an ADL. In the next section we formally define the failure behavior applied in our approach and describe how it can be manually modeled from an architecture without any computer-assisted task.

3.2 Failure Behavior

Reliability can be defined by either continuous or discrete-time events (as described in Section 2.3.1). In short, continuous-time is characterized by a failure rate in time units while discrete-time is given by a distribution of the number of non-failed executions in particular operating conditions.

In our view, continuous-time events assume that the software is always being utilized and a failure reflects a point in time when the system breaks down. However, some software systems are not used so intensively and only receive a few requests per day or during a period of time. The reliability of these systems should be determined through the number of invocations and successful responses instead of measuring the time that the system is idle waiting for requests.

With this in mind, our approach considers reliability to be a discrete-time unit where a failure may occur while a component is processing the required service or during the control transfer between two different components. Moreover, reliability is specified in terms of a percentage, denoting the number of successful resolved requests over the total number of requests performed by that specific module. For instance, if a module has eighty percent (80%) of reliability, it means that eight (8) out of ten (10) requests are well performed and the other two (2) fail for some reason, such as malformed input or another source of failure, including hardware and software failures.

Although there are no formal standards to obtain information about reliability data, there are some empirical means suggested by the literature [Gokhale, 2007; Goševa-Popstojanova & Trivedi, 2001]. More specifically, during the design phase an architect may consult the commercial entities that developed Commercial Off-The-Shelf (COTS) components. If the architecture encompasses non-COTS components, the failure data can be estimated from expert knowledge or through historical data for components developed in house. Regarding the already developed or deployed systems, the probability of failure might be extracted from the running system [Casanova *et al.*, 2011].

3.3 Combining Architecture with the Failure Behavior

The literature on reliability prediction proposes different modeling techniques to combine the architectural model with the failure behavior. Previously in Section 2.3.2, we explained their differences and with this in mind, we opted for the technique that offers the most accurate results: the composite approach of the state-based model.

The composite approach [Cheung, 1980] encompasses the generation from an architecture of an absorbing Discrete-Time Markov Chain (DTMC) with two final states s_C and s_F, that represent the correct and the failure outcome, respectively. Each state s_i represents a component of the software architecture and a directed branch $T_{i,j}$ represents a possible transfer of control from state s_i to s_j. In addition, R_i denotes the reliability of the state s_i. Since, we assume that every component can fail, each state s_i has a direct edge to the absorbing failure state s_F denoted by $T_{i,F}$. The transition probability to the s_F state is given by $(1-R_i)$

which represents the occurrence of an error in the execution of the component represented by the state s_i. The original transition probability between states s_i and s_j is modified to accommodate reliability and it is calculated as $R_i \cdot T_{i,j}$, representing the probability that state s_i executes correctly and the control is transferred to the component represented by the state s_j. Let the transition matrix be P where $P(i,j)$ represents the probability of transition from state s_i to state s_j in the Markov process.

$$P = \begin{bmatrix} I & 0 \\ C & Q \end{bmatrix}$$

The transition matrix P encompasses the identity matrix I, the zero matrix 0 and the matrices C and Q. The C matrix is the size of $(n \times 2)$ and holds the transition probabilities for the failure states. On the other hand, Q is an $(n \times n)$ matrix holding the transition probabilities among the states.

$$C = \begin{array}{c} \\ s_1 \\ s_2 \\ \vdots \\ s_{n-1} \\ s_n \end{array} \begin{array}{c} \begin{array}{cc} s_C & s_F \end{array} \\ \begin{pmatrix} 0 & T_{1,F} \\ 0 & T_{2,F} \\ \vdots & \vdots \\ 0 & T_{n-1,F} \\ R_n & T_{n,F} \end{pmatrix} \end{array}$$

$$Q = \begin{array}{c} \\ s_1 \\ s_2 \\ \vdots \\ s_n \end{array} \begin{array}{c} \begin{array}{cccc} s_1 & s_2 & \dots & s_n \end{array} \\ \begin{pmatrix} R_1 \cdot T_{1,1} & R_1 \cdot T_{1,2} & \dots & R_1 \cdot T_{1,n} \\ R_2 \cdot T_{2,1} & R_2 \cdot T_{2,2} & \dots & R_2 \cdot T_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ R_n \cdot T_{n,1} & R_n \cdot T_{n,2} & \dots & R_n \cdot T_{n,n} \end{pmatrix} \end{array}$$

The probability of starting in state i and entering in one of the absorbing states $j \in \{s_C, s_F\}$ is given by $P(i,j)$. To calculate the reliability of the system R_{sys}

we first need to solve the fundamental matrix given by M to find the computed value of the reliability after traversing the Markov chain.

$$\begin{aligned} M &= (I - Q)^{-1} \\ R_{sys} &= M(i,j) \cdot R_j \end{aligned}$$

The next section clarifies the previously outlined formulas through a demonstrative example.

3.4 Demonstration Example

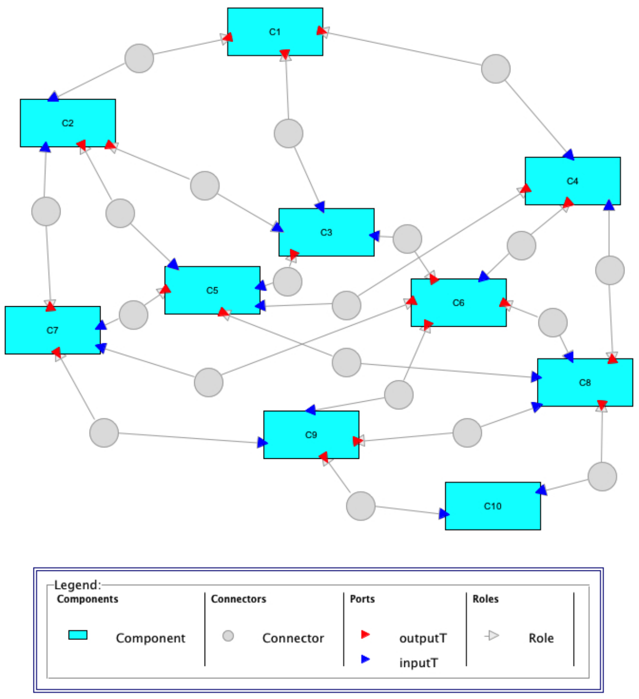
To illustrate the process of combining the architecture with the failure behavior we demonstrate analytically how to solve the Discrete-Time Markov Chain (DTMC) modeled from a software architecture adapted from different reliability studies [Cheung, 1980; Gokhale & Trivedi, 2002; Lo *et al.*, 2005].

The software architecture shown in Figure 3.1(a) is combined with the failure behavior to form an absorbing Markov chain depicting a state-based model as illustrated in Figure 3.1(b).

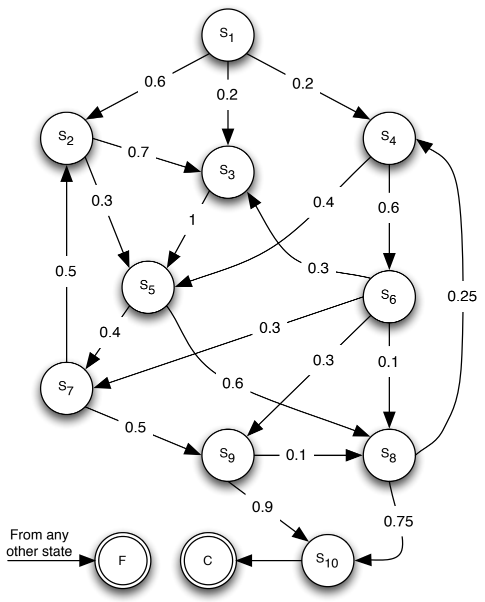
Figure 3.1(b) shows two absorbing states, F denoting the failure behavior receiving a direct edge from any other state in the system and C representing the correct behavior from the starting state s_1 until the last state is reached s_{10}.

The reliability and transition probabilities used in this example are merely illustrative. Table 5.2 shows the reliability values for each module and the transition probabilities, also known as usage profile, are given in Table 3.2.

To solve the DTMC we first need to determine the transition probability P.



(a) Software architecture described in Acme



(b) Markov reliability model

Figure 3.1: Software architecture (a) and its model (b)

Table 3.1: Component reliabilities

$R_1 = 0.99$	$R_6 = 0.95$
$R_2 = 0.98$	$R_7 = 0.98$
$R_3 = 0.99$	$R_8 = 0.96$
$R_4 = 0.96$	$R_9 = 0.97$
$R_5 = 0.98$	$R_{10} = 0.99$

Table 3.2: Transition probabilities (*i.e.*, usage profile)

$T_{1,2} = 0.6$	$T_{3,5} = 1.0$	$T_{6,3} = 0.3$	$T_{7,9} = 0.5$
$T_{1,3} = 0.2$	$T_{4,5} = 0.4$	$T_{6,7} = 0.3$	$T_{8,4} = 0.25$
$T_{1,4} = 0.2$	$T_{4,6} = 0.6$	$T_{6,8} = 0.1$	$T_{8,10} = 0.75$
$T_{2,3} = 0.7$	$T_{5,7} = 0.4$	$T_{6,9} = 0.3$	$T_{9,8} = 0.1$
$T_{2,5} = 0.3$	$T_{5,8} = 0.6$	$T_{7,2} = 0.5$	$T_{9,10} = 0.9$

$$P = \begin{array}{c} \\ s_C \\ s_F \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \\ s_7 \\ s_8 \\ s_9 \\ s_{10} \end{array} \begin{array}{c} \begin{array}{cccccccccccc} s_C & s_F & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 & s_9 & s_{10} \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.01 & 0 & 0.594 & 0.198 & 0.198 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.02 & 0 & 0 & 0.686 & 0 & 0.294 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.01 & 0 & 0 & 0 & 0 & 0.99 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.04 & 0 & 0 & 0 & 0 & 0.384 & 0.576 & 0 & 0 & 0 & 0 \\ 0 & 0.02 & 0 & 0 & 0 & 0 & 0 & 0 & 0.392 & 0.588 & 0 & 0 \\ 0 & 0.05 & 0 & 0 & 0.285 & 0 & 0 & 0 & 0.285 & 0.095 & 0.285 & 0 \\ 0 & 0.02 & 0 & 0.49 & 0 & 0 & 0 & 0 & 0 & 0 & 0.49 & 0 \\ 0 & 0.04 & 0 & 0 & 0 & 0.24 & 0 & 0 & 0 & 0 & 0 & 0.72 \\ 0 & 0.03 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.097 & 0 & 0.873 \\ 0.99 & 0.01 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

To calculate the system reliability in an analytical fashion, it is necessary to determine the Q matrix and compute the fundamental matrix as shown below.

$$Q = \begin{array}{c|cccccccccc} & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 & s_9 & s_{10} \\ s_1 & 0 & 0.594 & 0.198 & 0.198 & 0 & 0 & 0 & 0 & 0 & 0 \\ s_2 & 0 & 0 & 0.686 & 0 & 0.294 & 0 & 0 & 0 & 0 & 0 \\ s_3 & 0 & 0 & 0 & 0 & 0.99 & 0 & 0 & 0 & 0 & 0 \\ s_4 & 0 & 0 & 0 & 0 & 0.384 & 0.576 & 0 & 0 & 0 & 0 \\ s_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0.392 & 0.588 & 0 & 0 \\ s_6 & 0 & 0 & 0.285 & 0 & 0 & 0 & 0.285 & 0.095 & 0.285 & 0 \\ s_7 & 0 & 0.49 & 0 & 0 & 0 & 0 & 0 & 0 & 0.49 & 0 \\ s_8 & 0 & 0 & 0 & 0.24 & 0 & 0 & 0 & 0 & 0 & 0.72 \\ s_9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.097 & 0 & 0.873 \\ s_{10} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$M = (I - Q)^{-1} =$$

$$= \begin{bmatrix} 1 & 0.865 & 0.855 & 0.387 & 1.249 & 0.223 & 0.55 & 0.788 & 0.334 & 0.8598 \\ 0 & 1.275 & 0.907 & 0.2 & 1.35 & 0.115 & 0.562 & 0.835 & 0.308 & 0.87 \\ 0 & 0.28 & 1.225 & 0.203 & 1.374 & 0.117 & 0.572 & 0.849 & 0.313 & 0.885 \\ 0 & 0.262 & 0.368 & 1.15 & 0.883 & 0.662 & 0.535 & 0.626 & 0.451 & 0.844 \\ 0 & 0.283 & 0.228 & 0.205 & 1.388 & 0.118 & 0.577 & 0.858 & 0.317 & 0.894 \\ 0 & 0.266 & 0.488 & 0.123 & 0.609 & 1.071 & 0.544 & 0.515 & 0.571 & 0.87 \\ 0 & 0.628 & 0.449 & 0.111 & 0.672 & 0.064 & 1.281 & 0.463 & 0.646 & 0.898 \\ 0 & 0.062 & 0.088 & 0.276 & 0.212 & 0.159 & 0.128 & 1.15 & 0.108 & 0.922 \\ 0 & 0.006 & 0.008 & 0.026 & 0.02 & 0.015 & 0.012 & 0.111 & 1.01 & 0.962 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

To conclude, the system reliability is determined by solving the Markov model and also by taking into account the reliability of the last state, as shown in the following formula. Moreover, the prediction from the shown example yields approximate 85% (eighty five percent) of overall reliability.

$$R_{sys} = M(1, 10) \cdot R_{10} = 0.8598 \cdot 0.99 \approx 0.8512 = 85.12\%$$

The purpose of this demonstration was to show the analytic prediction of reliability and how stochastic modeling can be used to evaluate quality attributes from a software architecture. Although our main goal is to propose automated methods to predict these quality attributes, we believe it is important, as a starting point, to understand where these formalisms and techniques came from and how they are applied. The next section formally specifies the translation procedure between a software architecture and a Discrete-Time Markov Chain (DTMC).

3.5 The Translation Process

An automated process to predict reliability from a software architecture requires a translation from the architectural model to the corresponding mathematical formalism. To this end, we propose a formal translation process in which the architectural model (A) is related to the generation of the Discrete-Time Markov Chain (M). This relation is specified in Z by linking each member of the architecture to exactly one member of the DTMC as $A \rightarrow M$. Specifically we define a mapping from each architectural element of an Architectural Description Language (ADL) (*e.g.*, components, connectors, properties and their relations) to the respective states of the Deterministic-Time Markov Chain (DTMC). This mapping aims to achieve an automated generation of a DTMC from an architectural description.

Following this, we describe the translation process along with its required properties.

3.5.1 Initial state

The stochastic model requires the specification of the initial state where the control flow begins. This involves annotating the architectural model with a mandatory property called ‘EntryPoint’ which states the starting component $c_i \in C$ and will be mapped as the initial state in the DTMC.

$$Prop = (EntryPoint, String, Name(c_i)) \mapsto \overline{s}$$

3.5.2 Components

The translation of each component is performed in accordance with the following guidelines:

- Each component maps to a state

$$c_i \in C \mapsto s_i \in S$$

- The reliability of a component is defined as the probability that component c_i will carry out a task successfully with no failures. A non-failed task occurs when the component processes data received by an input port and sends a response over the output port.

$$Rc(c_i) = Pr\{c_i \in C\,;\, ip_j \in IP(c_i)\,;\, op_k \in OP(c_i) :\ op_k \text{ produces a correct output} \mid ip_j \text{ received an input}\} \in [0,1]$$

  Rc is expressed in the architectural model through a component property, by mapping to the probability of successfully transiting (P_R) from state s_i to s_{i+1} and also to the probability of transition to a failure absorbing state s_F:

$$\begin{aligned} Prop = (reliability, float, Rc) &\mapsto P_R(s_i, s_{i+1}) = Rc \qquad \wedge \\ Prop = (reliability, float, Rc) &\mapsto P(s_i, s_F) = 1 - Rc \end{aligned}$$

- When specified, a representation expresses the internal behavior of a component by providing a new sub-system:

$$Rep = (C',\ Con',\ Att',\ Prop') \mapsto s_i = (S', \overline{s}', P', L')$$

3.5.3 Connectors

Each connector $con_i \in Con$ is responsible for the communication between different components and plays an important role in the system reliability and hence, its translation is performed as follows:

- The probability of transiting between two components is specified as an attachment that sends data knowing that another attachment has already received data from a component.

$$T(con_i) = Pr\{att, att' \in Att\,;\, c, c' \in C : att' \text{ communicates data to } c' \mid att \text{ receives data from } c\ \} \in [0,1]$$

The transition probability (P_T) is expressed through the property

$$Prop = (transitionP, float, T) \mapsto P_T(s_i, s_{i+1}) = T$$

To calculate the probability of transiting from state s_i to s_{i+1} we use the following formula:

$$P(s_i, s_{i+1}) = P_T(s_i, s_{i+1}) * P_R(s_i, s_{i+1})$$

3.5.4 Constraints

Before the generated model can be considered valid, it must conform to the following constraints specified in the Z notation:

No dangling ports or roles Every input or output port of a component must be attached to a connector role and the same applies conversely. The system is not valid if there are any dangling ports or roles.

$$\begin{array}{ll} & \forall\, ip \in IP(c) \mid \exists\, c \in C; r \in R(con); con \in Con \bullet ((c, ip), (con, r)) \in Att \\ \wedge & \forall\, op \in OP(c) \mid \exists\, c \in C; r \in R(con); con \in Con \bullet ((c, op), (con, r)) \in Att \\ \wedge & \forall\, r \in R(con) \mid \exists\, con \in Con; c \in C; ip \in IP(c); op \in OP(c) \bullet \\ & \qquad\qquad ((c, ip \vee op), (con, r)) \in Att \end{array}$$

Output transitions sum to 1.0 For every output port, there exists a set with at least one connection to a target component. For all the elements within that set, their transition probabilities must add up to 1.

$$\forall\, op \in OP \mid \exists\, con_i \in Con; T \in Props\,(con_i) \bullet \sum_i T_{con_i} = 1.0$$

This section describes the mapping from an architectural description to a mathematical model. We specified this mapping formally to allow an unambiguous translation from an ADL to a DTMC and support its future extension to other quality attributes.

Software applications use complex interactions to share data among components, sometimes expressing common behaviors or patterns. Bearing this in mind, the next section proposes the integration of different architectural styles within our approach.

3.6 Modeling Architectural Styles

Architectural styles, also known as architectural patterns, are well-documented solutions to frequently recurring problems which may display known quality attributes [Bass *et al.*, 1998; Garlan & Shaw, 1994]. Modeling these styles in an early phase allow architects to make decisions that achieve particular quality goals and assess if certain architectural choices are made to the detriment of others. In this section, we outline the method that can be employed to model these styles by specifying the required modifications to the original Discrete-Time Markov Chain (DTMC).

3.6.1 Batch-sequential / Pipe-and-Filter

The batch-sequential and pipe-and-filter style share the same ideology in which the components in the architecture are executed in a sequential order. Their difference resides in the fact that in the batch-sequential style only one component executes at any instance of time, while in the pipe-and-filter an output may be produced before its inputs are fully consumed. To clarify this difference, an example of a batch-sequential style is for instance the extraction of a ZIP file that will be processed only after its extraction has been concluded, otherwise it could lead to a corrupted file. On the other hand, an example of the pipe-and-filter style is a UNIX command that allows data to be filtered while a file is being read (*e.g.*, *cat file.txt* | *grep word*). From the provided example, the *grep* program executes by reading the output of the *cat* command without requiring it to be completed.

Batch-sequential and pipe-and-filter styles differ in the control flow of data, but they share the same mathematical model. Their modeling approach is illustrated in Figure 3.2 where they display a sequential order of execution between the components. Note that $T1, 2$ must equal 1 to respect constraints in Section 3.5.4

while at the same time keeping coherence with matrix Q in Section 3.3.

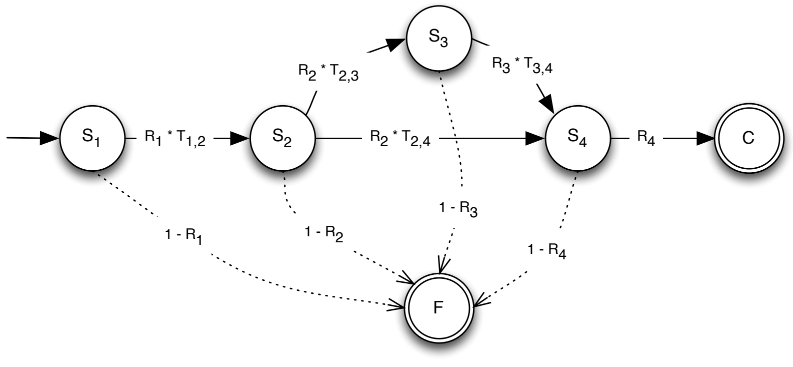


Figure 3.2: Batch-sequential style state model

3.6.2 Parallel

The parallel style is commonly used to model systems that carry out concurrent executions, usually aiming to improve performance. In this style, the work is partitioned and each component works on a small sub-task to complete a larger task.

This style can be modeled, as shown in Figure 3.3, where architectural components that represent a single execution are mapped to States s_1 and s_3, in the same way as the batch-sequential style. The inherent concurrent executions in the parallel style are mapped to different states, as shown by states $s_{21}, s_{22}, \dots, s_{2k}$. These states are wrapped in the s_2 state, which is responsible for the synchronization process of when a transition to these states occurs and when the concurrent executions of all the parallel states are completed.

In case of failure by one of the concurrent states, the computation of the sub-task will not be completed and the system will go into a failed state. As a result, the execution of the state s_2 will only be correct if all the sub-tasks are equally correct. This leads to the following formula to calculate the reliability of the state responsible for the coordination of the concurrent executions.

$$R_i = \prod_{j=1}^{k} R_{ij}$$

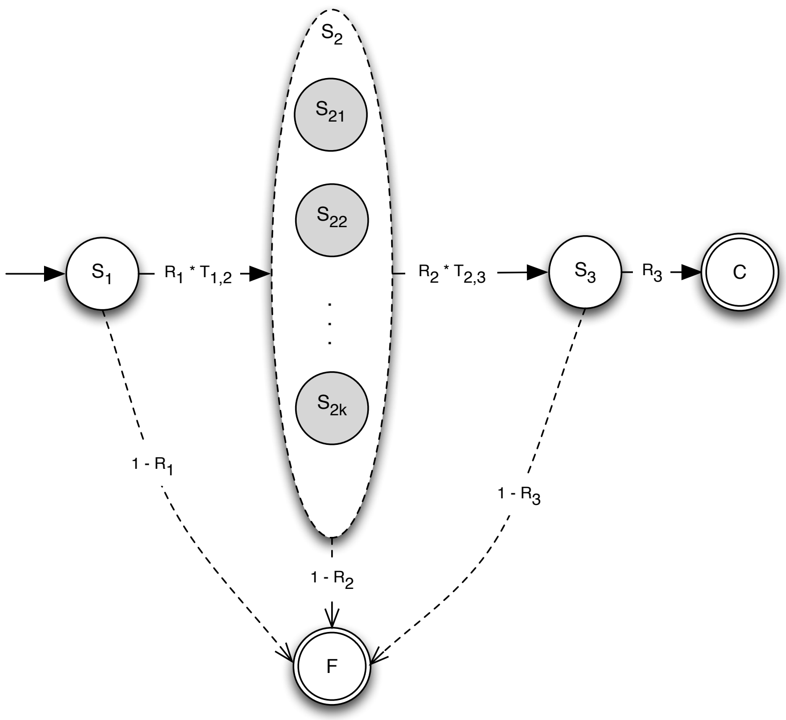


Figure 3.3: State model of the parallel style

3.6.3 Fault-tolerance

A fault-tolerance style can be applied to a system to obtain higher reliability or fewer failures in a specified time period. This style is composed of a set of components that seek to compensate for the failures of each other. More specifically, only one component performs computation – the active component. When it fails, one of the redundant components takes over and becomes the active one. The system only enters in a failed state, when all the components in the fault-tolerant set fail. Our approach supports different reliability values for each of the components in the set, as they may involve different data structures or algorithms to improve the system reliability.

The state model of this style is illustrated by Figure 3.4, where the fault-tolerant set is shown wrapped in the state s_2 is shown. The active state is depicted with a gray background and the redundant states with a white one.

The execution of this type of model only fails when all the active and re-

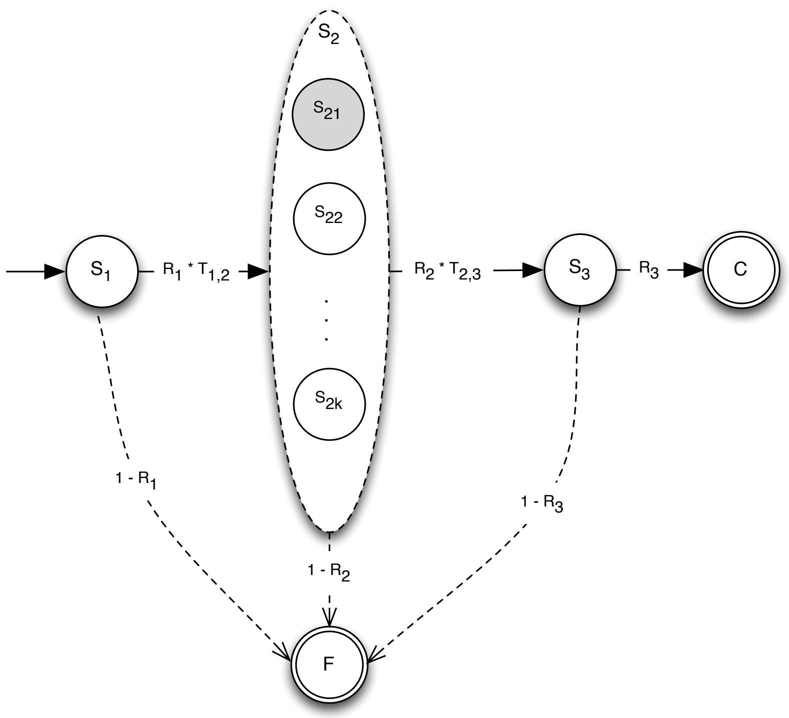


Figure 3.4: Fault-tolerance style state model

dundant components fail. The following formula is proposed to calculate the reliability of this style and is supported by Figure 3.4.

$$R_i = 1 - \prod_{j=1}^{k}(1 - R_{ij})$$

3.6.4 Call-and-return

In the call-and-return style, a caller component may request services provided by other called components. When the services are requested, the caller component holds its execution, until the called one has fulfilled the request. After that, the caller component resumes its execution where it left off. This style is often used for remote procedure calls and it can be translated to the state space model as follows: a state represents an execution of a component and a transition takes place when the execution in each state has been completed or the execution encounters a service request that temporarily transfers the control to the called component.

Figure 3.5 depicts a call-and-return style, where state s_1 is the caller and s_2 is the called component. After calling state s_2, the caller state will transfer the control to state s_3, which, if everything went as expected, means the control will be transferred to the absorbing component s_C. As can be seen in Figure 3.5, the transition $T_{1,2}$ does not take account of the reliability of the caller component, because s_1 will be visited only once before transferring the control to state s_3 regardless of the number of times the state s_2 is called [Wang *et al.*, 2006].

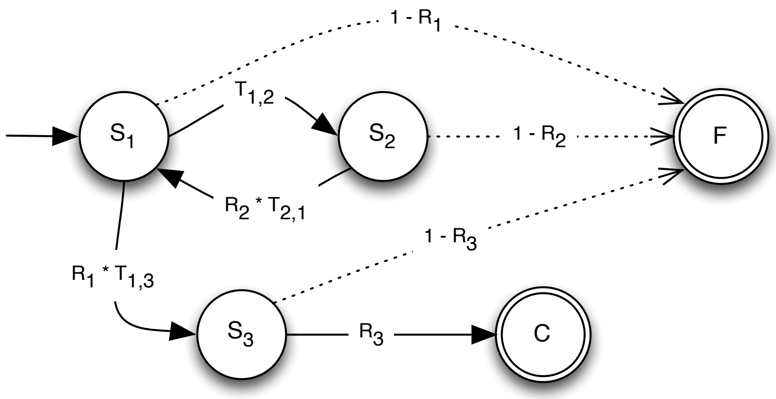


Figure 3.5: Call-and-return style state model

In summary, this section has described how to incorporate architectural styles in our modeling approach. As a result, architects can model software with heterogeneous styles and make use of different execution orders beside the sequential one. The process of translating architectural styles to the model can be accomplished by annotating components and connectors with relevant properties.

The next section outlines the process that makes possible the automated reliability prediction from an ADL description by examining the required tools and methods.

3.7 Automated Prediction

Predicting reliability from a software architecture is a process that encompasses several steps, such as the identification of the software architecture and its constituents, notations holding non-functional properties and their translation to a stochastic model. This model includes the failure behavior encoded in the architecture that makes it possible to determine whether the quality requirements are being met by the proposed architecture.

Several research surveys have found shortcomings in the current approaches to reliability prediction, such as the lack of support for tools and for variability [Gokhale, 2007; Immonen & Niemelä, 2008; Pengoria *et al.*, 2009]. One of the main reasons is that current approaches rely on manual activities which are usually error-prone and time-consuming tasks. With this in mind, Martens *et al.* [2010] adopted an approach that quantitatively predicted performance, reliability and costs from a software architecture specifically described in Palladio [Martens *et al.*, 2010], a custom specific language designed by Martens *et al.* to describe a system. However, their work did not take into account different architectural styles, nor the kinds of formalisms required to extend the translation procedure to other quality attributes or Architecture Description Languages (ADLs).

As a result of the shortcomings highlighted by current surveys and from the lack of methods or techniques proposed in the research community, our approach addresses the question of reliability prediction from a software architecture that operates in an automated fashion. For this purpose, we formally annotate the translation procedure by making it possible to parse and analyze an ADL by complying with the ISO/IEC/IEEE 42010 Standard [ISO/IEC/IEEE, 2011]. Furthermore, we test the effectiveness of our approach by applying it to the Acme ADL [Garlan *et al.*, 1997]. The reason for this choice rather than the other ADLs examined in Section 2.1.3 is due to the fact that Acme is a general purpose language and not domain-specific like the others [Taylor *et al.*, 2009]. In addition, the development team of Acme, the Software Engineering Institute (SEI) in Carnegie Mellon University (CMU), have set up a software library, AcmeLib, that allows Acme models to be manipulated by third-party applications.

After parsing the architectural models, we generated a Discrete-Time Markov Chain (DTMC) in the Prism language [Kwiatkowska *et al.*, 2009]. The generated formalism is then resolved by the Prism Model Checker tool [Kwiatkowska *et al.*, 2011] which provides a quantitative result for the system reliability. Although other probabilistic model checking tools could be applied, we selected Prism since it is a free, open-source tool with a vast documentation and support for discrete-time Markov chains. A comparison between Prism and other tools is provided in Section 2.3.2.

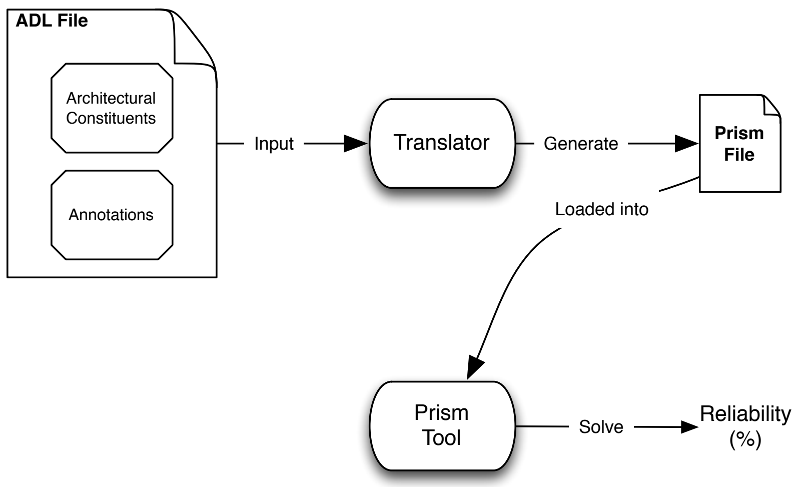


Figure 3.6: Automated Reliability Prediction

The automated prediction process outlined above is illustrated in Figure 3.6. We parsed the ADL file responsible for holding information about the topology of the system, its architectural constituents and annotations which describe non-functional properties. Following this, we combined the architectural model with the failure behavior into a Discrete-Time Markov Chain (DTMC), which is described as a high-level language in the Prism File. Finally, Prism tool solved the stochastic model by taking into account the PCTL rules and providing a quantitative result about the reliability of the overall system.

3.8 Threats for Validity

The approach proposed in this chapter acts accordingly to the following assumptions:

Every software component can fail. Each module that is mapped from the architecture to the mathematical model has a direct edge to the absorbing state s_F, which is weighted by its probability of failing (*i.e.*, one minus the reliability assigned to the component).

The failures are independent of the software components. Components in a software system can be viewed as logically independent modules, which can be developed and tested independently from each other [Gokhale & Trivedi, 2002; Lo *et al.*, 2005].

The transfer of control among the modules follows a Markov Process. The transition probability from one component to another is determined through the product of the reliability of that component with the estimated usage profile of the system. This means that the control transition is independent of the past history of the system and only depends on the current state, following the memoryless property of a Markov chain [Grinstead, Charles M. and Snell, 2006].

System reliability is the probability of reaching the state *C*. The computation of the system reliability is performed through the probability of transit between all the components in the system and reaching the absorbing state s_C, by displaying the correct behavior of the system or the probability of every component in the system being failure-free.

These assumptions are due to the stochastic model used and may vary if other modeling approach is applied. The next section outlines other studies and research contributions that are related or serve as a basis to this work.

3.9 Related Work

Several studies address the reliability assessment from a software architecture description [Brosch *et al.*, 2011; Cortellessa, 2002; Reussner & Heinz W., 2003; Yacoub *et al.*, 2000], among the firsts to propose architecture reliability modelling using Markov chains was Cheung [Cheung, 1980] and several surveys were presented since then [Gokhale, 2007; Goševa-Popstojanova & Trivedi, 2001; Immonen & Niemelä, 2008; Pengoria *et al.*, 2009].

Above methods are theoretical mathematical models to assess the reliability in an early software development stage that are accurate enough to be applied to real case studies. Popstojanova *et al.* [Goseva-Popstojanova *et al.*, 2005] studied and tested the test suite of the C compiler to prove the adequacy, applicability and accuracy of software reliability models. The results obtained show that the actual reliability differs only by less than 3% from the theoretical methods, proving that both the composite and hierarchical models are very accurate and applicable for real case studies.

Over the years, common patterns of structural organization of components and connectors have been identified and documented [Garlan & Shaw, 1994]. The so-called architectural styles are commonly used in any architecture, but they impose constraints in reliability assessment: each architectural style maps to a different state-space model and it must be extended to reflect some architectural choices, such as concurrency or fault-tolerance. Abd-allah [Abd-Allah, 1997] identified the issues of reliability assessment of architectural styles using reliability block diagrams and Wang *et al.* [Wang *et al.*, 2006] described the process of mapping a limited number of architectural styles to state space models for reliability analysis. Only few research studies address the reliability analysis on architectural styles. More interest is needed on this topic to analyze the reliability on important styles that were not considered before, such as event-based or black-board repository.

Martens *et al.* [Martens *et al.*, 2010] present an approach to quantitatively predict the performance, reliability and cost of a software architecture. Their approach supports a multi-criteria genetic algorithm to find the best trade-off between those quality attributes. Regarding reliability analysis, they performed two types of analysis: transform the software architecture into an absorbing discrete-time Markov chain and a reliability simulation to derive the probability of failure on demand. Their work differs from ours in the following aspects: 1) they use a custom specific language while we use a general purpose ADL; 2) they do not take into consideration the different architectural styles that may be used; and 3) we include a formal notation to extend the current approach to other quality attributes.

3.10 Summary

The process of automating the prediction of reliability from a software architecture is only possible on account of the wide range of techniques available to describe an architecture and the methods employed to calculate reliability as a quality attribute. Our approach performs a reliability prediction by leveraging the implicit formalisms in ADLs to extract the required information.

The proposed approach encompasses a modeling approach that is time-agnostic, and relies on the probability of a component correctly delivering a request. In addition, our method envisages a formal description of the translation procedure to the stochastic model of choice (*i.e.*, a Discrete-Time Markov Chain (DTMC)) which can be extended to encompass different modeling approaches, as well as other quality attributes.

This approach can be applied within the software development life-cycle. In particular in the design stage, architects can perform reliability predictions to assure that the designed artifact conforms to the quality requirements outlined by the stakeholders. Moreover, this approach can also be applied in an evolutionary stage to identify reliability bottlenecks, test different architectural alternatives and assure the achievement of the desired quality attributes. Although in an evolutionary stage, it can be assumed that there is a deployed system that can be instrumented to obtain data about the usage profile and components' reliability, the same does not apply in an early phase such as design. In this phase, information is scarce about implementation details and even more about failure rates and usage profiles. Thus, to obtain failure data in the design phase, architects can use Hidden Markov Chains [Cheung *et al.*, 2008], consult COTS commercial entities, estimate failure data from expert knowledge or through historical data for components developed in house. Although a reliability figure in such an early phase may not be certain, this type of analysis helps architects to understand issues in the architecture and identify modules and interconnections that deserve attention when making future improvements.

In short, the automated generation of a mathematical model from an ADL saves software architects the effort of having to build it manually by providing a correct and error-free formal model. In addition, this approach provides the means for architects to test, assure and experiment with different architectural alternatives. More specifically, an architect can vary the number of components, their arrangement and their interconnections, as well as changing styles and modifying system reliability and usage profile values. As a result, our approach provides assurances for architects on whether the designed artifact meets the requirements established by the stakeholders.

Chapter 4

Automated Sensitivity Analysis

Software architecture is a fundamental activity of software development, in which designers are able to reason about a system's structure and properties at a high level of abstraction, before any design and implementation is carried out. This means that software architecture facilitates early decisions on systemic properties such as reliability, maintainability, and performance. Although it is very useful in the early stages of development, a software architecture also provides value as the software evolves and new versions are planned and developed. The redesigning and modifications can be put into effect at the architectural level so that different architectural alternatives and what-if testing scenarios can be compared. At this level of abstraction, architects can analyze the system, test alternatives and avoid the costs involved in the implementation and deployment stages.

There have been a number of techniques to analyze the outcome of the design phase, either in the form of a software architecture or a set of textual documents, employed by the software engineering community in the last few years [Gokhale & Trivedi, 2002; Lo *et al.*, 2005]. These techniques give architects the ability to quantitatively or qualitatively assess their designs and identify architectural elements that require improvement. However, most of the suggested methods entail manual activities which might make sense while dealing with textual documents, but formally described architectures in a digital format open up the opportunity to use automated methods. In light of this, we have outlined an automated approach that performs a thorough reliability analysis for an architectural design. Our analysis is capable of identifying weak architectural features and interconnections that are not operating well. Finally, we established a ranking system to order architectural elements according to their effect on the system reliability.

Our goal is to provide architects with the means to analyze their architectural designs and find alternatives to increase the overall system reliability. With this end in view, our analysis investigates which components affect the system the most (4.1). Following this, we analyze the interconnections between the software components to evaluate how the system is used and which architectural paths should be improved (4.2). The proposed approach also includes a ranking system to sort out architectural elements in terms of their reliability (4.3). To show the applicability of our method, we employ the proposed analysis method through an example architecture (4.4). Then, we examine the required techniques that automate the proposed approach and allow architects to analyze their own designs without the need for manual assessment (4.5). Finally, we present the related work to the one outlined in this chapter (4.6).

4.1 Variation in Component Reliability

System reliability depends to a great extent on the individual component reliability but also on how the system is used and organized; *i.e.*, a system can be highly dependable, although it consists of unreliable components. The reason for this is that these unreliable components may not be used so often and hence do not affect the overall system reliability. Moreover, they may be structured in such a way that unreliable components "check" each other for errors, such as whenever replication is used (e.g. TMR). The identification and correction of these unreliable components is an important task that must be carried out early in the development cycle, since improvements and corrections made at later stages involve high costs and delays.

The identification of components that act as reliability bottlenecks is not a task of simply picking the components with the lowest reliability. On the contrary, the bottleneck may be a component that has a high reliability value, but owning to the high rate of usage increases the chance of triggering a software defect and leading to a failure. Thus, our approach involves conducting a sensitivity analysis on the reliability of every component to obtain data about their influence on the overall system reliability. On the basis of this information, the architect can concentrate on improving a particular component by inducting more testing and code inspections or even, by correcting bugs.

Reliability can be described as a “probability of failure-free operation” or the continuous delivery of a correct service belonging to the interval $R \in [0, 1]$. Generally, a sensitivity analysis applies a linear variation on reliability, which may be a shortcoming for the following reasons:

- The variation of reliability through a linear approach may exceed the range of the reliability interval. This is illustrated by the example in Equation 4.1, where R_i represents the reliability of component i. We carried out a linear variation on the reliability value, which showed that the result falls outside the interval of possible values for the reliability.

$$R_i = 0.99 \pm 10\% = \{0.891, 1.089\} \notin [\mathbf{0}, \mathbf{1}] \tag{4.1}$$

- A linear variation requires an increased effort from developers to improve the same percentage in systems that present higher reliability values than lower ones. More specifically, when two systems have 50% and 90% of overall reliability, their linear variation of 10% requires an improvement of 5% and 9%, respectively. However, in systems with 90% reliability it is much harder to identify and correct software defects than in a system with a lower reliability value. Thus, we consider that the linear variation does not reflect the actual improving effort, since increasing 10 % of reliability in a system is different from 10% on another.

To overcome these problems, we apply a logarithmic variation to the reliability of individual components. In short, this logarithmic variation makes system improvements more expensive as we approach 100% reliability. Its formula is outlined in Equation 4.2.

$$R = 1 - 10^{-x} \ \Leftrightarrow\ x = -\log_{10}(1 - R) \tag{4.2}$$

To show the applicability of the logarithmic variation, we varied by 10% a system presenting an overall reliability of 99%. Equation 4.3 exemplifies this variation, where R is the reliability and U represents the unreliability.

$$\begin{aligned}
R_i &= 0.99 \\
U_i &= 1 - R_i = 0.01 \\
x &= -\log_{10}(U_i) = -\log_{10}(0.01) = 2 \\
x &\pm 10\% = \{1.8, 2.2\} \\
R_i &= 1 - 10^{-x} = \begin{cases} R_i \; min = 1 - 10^{-1.8} = 0.9841 \\ R_i \; max = 1 - 10^{-2.2} = 0.9936 \end{cases}
\end{aligned} \tag{4.3}$$

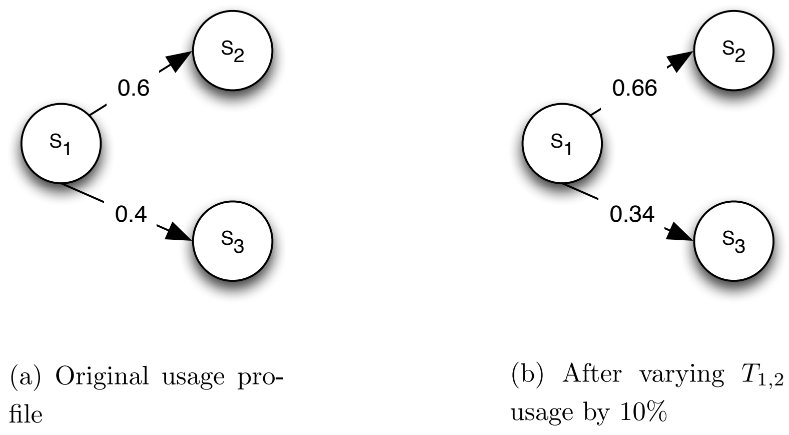
The above example shows that a component with 99% of reliability would vary within the range of 98.41% and 99.36%. However, the prediction method for reliability outlined in Chapter 3 also adopts the usage profile as a modeling attribute. The following section discusses the process of analyzing the system usage profile.

4.2 Analysis of the Usage Profile

Each user carries out different tasks in the system, which leads to distinct invocations of different methods or functions. The system usage profile can be defined as an estimation of how the system will be used and refers to the inter-components transition probability.

To analyze the system usage profile we propose to vary the usage of each path and observe its impact on the system reliability. After probing all the system paths, we were able to identify those that affect the system the most and suggest how they could be improved. The variation of usage profile must comply with a constraint: the sum of all the output transitions from a component must equal 1.0. In other words, if we vary an output connection from a component, then the other connections must also vary in the same order of magnitude. In view of this, Figure 4.1 shows an example of this variation in a system where component C_1 mapped as state S_1 has two output connections. In more detail, we increased the usage profile of the connection $T_{1,2}$ by 10% and inversely, we reduced the same amount in the connection $T_{1,3}$ as shown in Figure 4.1(b).

Although this constraint looks rather simple for systems having at most two connections, it becomes complex in systems where there are three or more in-



(a) Original usage profile

(b) After varying $T_{1,2}$ usage by 10%

Figure 4.1: Illustrative example of the usage profile variation in a system with two connections

terconnections between the components. In other words, the variation in each connection will depend on its relative usage or weight for the other ones.

To address this issue, Equation 4.4 was used to vary a specific output connection $T_{i,j}$ by obtaining the new transition probability $T'_{i,j}$. In more detail, if we increase the usage profile of a particular connection, we have to proportionally decrease the other output connections, denoted by $T_{i,k}$. Hence, after calculating the new transition probability $T'_{i,j}$ of the connection we want to vary, we have to calculate the new transitions for the other connections denoted by $T'_{i,k}$.

$$\begin{aligned} T'_{ij} &= T_{ij} + variation\% \\ \Delta t &= T_{ij} \cdot variation\% \\ T'_{ik} &= T_{ik} - \frac{\Delta t * T_{ik}}{\sum_{k \neq j}^{n} T_{ik}} \end{aligned} \tag{4.4}$$

As an illustrative example, Figure 4.2(a) depicts a system with three output connections from the same component. To demonstrate the applicability of our method, we made a variation of 10% in the usage profile in the transition $T_{1,2}$. The other transition probabilities were obtained by means of Equation 4.4 and the result is given in Figure 4.2(b).

After proposing methods to study the variation of sensitivity in different kinds of reliability and usage profiles, the next section proposes a technique to identify

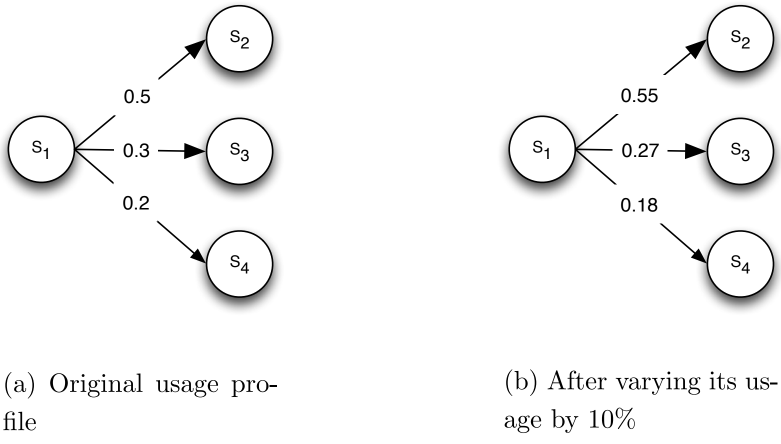

s1
s2
s3
s4
0.5
0.3
0.2
(a) Original usage profile
s1
s2
s3
s4
0.55
0.27
0.18
(b) After varying its usage by 10%

Figure 4.2: Illustrative example of usage profile variation in a system with more than two connections

weak architectural points which involves ranking their effects on the overall system reliability.

4.3 Analysis of ranking

The analysis of the usage profile and reliability are important to identify architectural problems as well as to find alternatives to the proposed design. For this reason, we employ a method to rank components and connectors in order to find out which ones influence the system the most.

The rank is obtained by calculating the derivative of each curve around the point where the variation is null (*i.e.*, variation = 0%), as shown by Figure 4.3. This derivative represents the slope of the curve between the imposed variation and its impact on the overall system reliability. As a result, if the slope presents values close to zero (0), it means that the impact of the variation is almost imperceptible in the overall system reliability. On the other hand, how higher the slope is, the higher is the impact of the variation on the overall system reliability.

At the end, we sort the derivatives of each curve to identify which ones have the highest impact and should be improved. This ranking can be performed for both reliability and usage profile analysis.

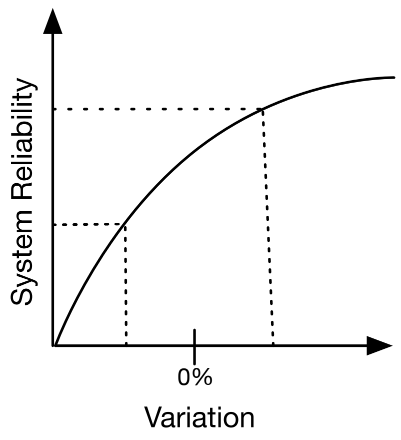


Figure 4.3: Derivative around where the variation is null

To clarify the proposed methods to analyze an architecture, we used the following demonstration to show their effectiveness and applicability.

4.4 Demonstration

An analysis of an architecture to determine its reliability includes varying the non-functional properties and seeks to identify problems and find possible solutions. In this section we aim to demonstrate the effectiveness by employing our method when applied to the example of architecture described in Chapter 3.4.

4.4.1 Analysis of Reliability

A sensitivity analysis examines key issues and determines which components are reliability bottlenecks. In more detail, we vary the reliability of each component and rank in accordance with their influence on the overall system. To demonstrate this procedure, we used the same architecture provided in Figure 3.1 as well as the same reliability and usage profile values.

Figure 4.4 depicts the variation of 10% of the reliability for each component in the system. The reliability variation is shown in the x-axis where the middle point corresponds to the null variation (*i.e.*, 0.0) and the y-axis represents the overall system reliability. We made a variation in each component to identify which elements have more effect on the system reliability.

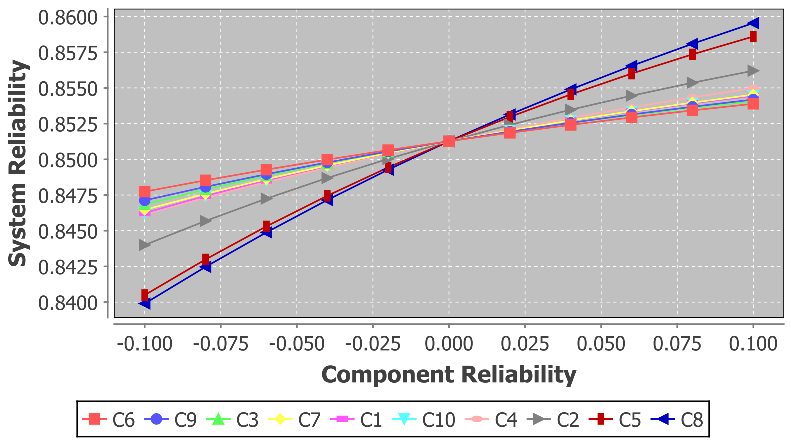


Figure 4.4: Sensitivity analysis with respect to reliability

After this, the components are ranked according to their variation slope to find out which influence the system the most, as shown in Table 4.1. More specifically,

Table 4.1: Results of the component reliability analysis

$\mathbf{C}_i$	$\mathbf{R}_i$	**Partial Derivative**
C8	0.96	0.096
C5	0.98	0.088
C2	0.98	0.059
C4	0.96	0.043
C10	0.99	0.039
C1	0.99	0.039
C7	0.98	0.039
C3	0.99	0.034
C9	0.97	0.034
C6	0.95	0.030

components $C8$ and $C5$ are on the top of the list, which shows that they have a greater effect on the overall system reliability and the architect must make improvements to these components to increase the overall system reliability.

4.4.2 Usage Profile Analysis

The analysis of the system usage profile is shown in Figure 4.5, with the three best and worst usage profile variations from the total of nineteen inter-component transitions. To support Figure 4.5, Table 4.2 lists the sorted ranks obtained from

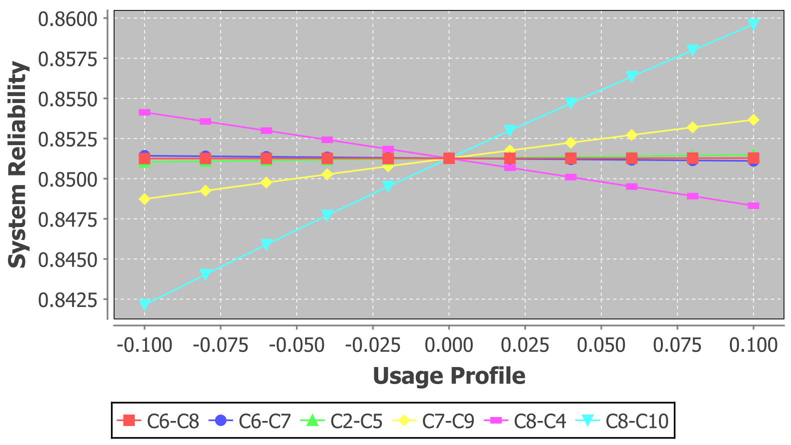


Figure 4.5: Sensitivity analysis with respect to the usage profile

the analysis. It can be concluded from the information that is presented that the inter-component transition from $C8$ to $C10$ is the one that has a greater impact on the overall system reliability. On the other hand, increasing the usage of the connection between component $C8$ to $C4$ will have a negative effect. This can be explained by the fact that the more we raise the usage profile of $C8 - C4$, the more the system will be used, increasing the chance of failing requests.

4.4.3 Making Structural Changes

The results of the sensitivity analysis carried out in the above sections reveal how possible architectural improvements can be made. On the basis of this information, we made some changes at the architectural level with the aim of improving the overall reliability. In particular, the components that act as reliability bottlenecks were identified as $C8$ and $C5$. The analysis of the usage profile suggested there could be an improvement in reliability through modifying the transitions in the $C8$ component by lowering the usage of $C8 - C4$ and increasing the load of the $C8 - C10$.

Table 4.2: Results of the usage profile analysis

C$_i$-C$_j$	**Partial Derivative**
C8-C10	0.087
C8-C4	0.029
C7-C9	0.024
C2-C5	0.002
C6-C7	0.001
C6-C8	1.5E-4

Thus, we propose an evolution of the architecture by making the following architectural changes:

- Reliability improvement of 10% on components $C5$ and $C8$.
- Usage profile variation of 10% on $C8 - C4$ and $C8 - C10$.
- Changes to the topology by adding one extra component ($C11$). This component replicates the functionality of $C8$ in order to reduce the connection from $C5 - C8$.

Figure 4.6 illustrates the new architecture that encompasses the above modifications and Table 4.3 shows the reliability values that are used for each component in the system.

Table 4.3: Reliability values of the new architecture

$R_1 = 0.99$	$R_6 = 0.95$
$R_2 = 0.98$	$R_7 = 0.98$
$R_3 = 0.99$	$R_8 = 0.971$
$R_4 = 0.96$	$R_9 = 0.97$
$R_5 = 0.986$	$R_{10} = 0.99$
	$R_{11} = 0.99$

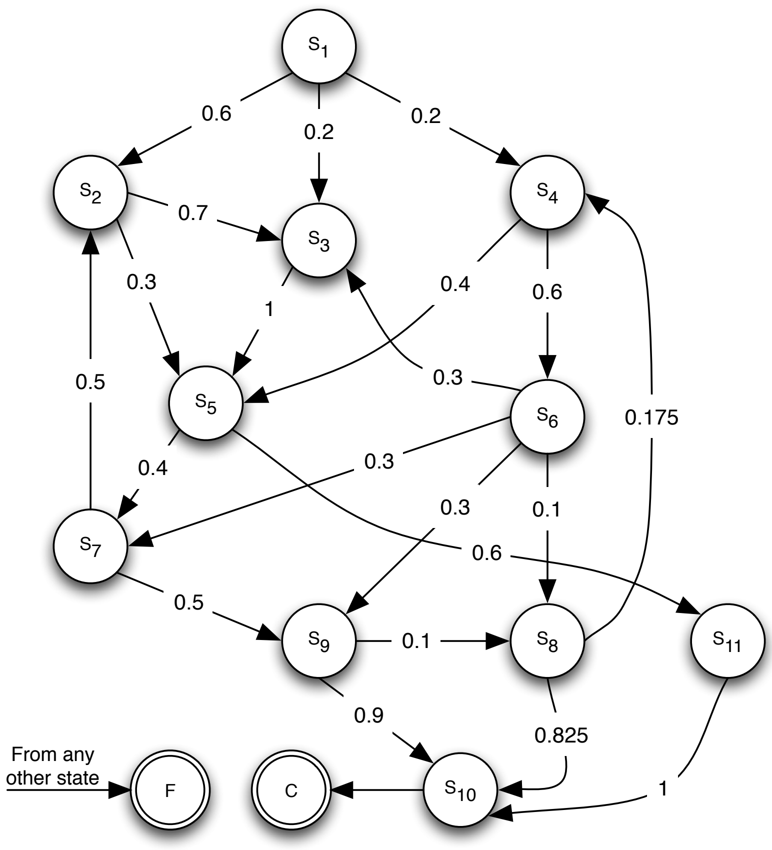


Figure 4.6: State model of the new architecture

The estimated system reliability before the changes was of 85.12% and after making the structural changes, the estimated system reliability rises to 90.23%, leading to a reduction in unreliability of about one third.

As a result, we showed that our approach contributes to the improvement of the overall system reliability by providing important insights into architectural modification and evolution. The following section discusses how to automate the analysis procedure to reduce the manual effort.

4.5 Automated Analysis

Just as in Chapter 3 we employed an automated method to predict the reliability from a software architecture, in this chapter we also intend to perform a reliability analysis from a system in an automated fashion. This was undertaken by employing the methods proposed in above sub-sections to vary the usage profile and the reliability of the software constituents. Our main goal was to provide the means for architects to test their system with a minimum effort and identify bottlenecks and constituents elements that required improvements.

To automate the reliability analysis, we generated a stochastic model from the software architecture and conducted a sensitivity analysis on its reliability and usage profile parameters. Our approach varies these values directly in the stochastic model and determines how the results change as the inputs are varied. This makes it possible to decide which components or interconnections are having a negative influence on the system and should be improved.

4.6 Related Work

Barais *et al.* [Barais *et al.*, 2008] studied diverse state-of-the-art approaches to evolve software architectures. They conclude that even though there are several ADLs that enable architects to specify their software systems, most of them do not provide means to evolve the architecture. Our approach adds value to these ADLs by assessing the reliability at the software architecture level. In addition, our approach is detached from the architecture, allowing it to be applied to any available ADL that complies with the ISO/IEC/IEEE 42010 [ISO/IEC/IEEE, 2011] standard.

Gokhale *et al.* [Gokhale & Trivedi, 2002] and Lo *et al.* [Lo *et al.*, 2005] perform a sensitivity analysis on the reliability of a software architecture by varying the expected usage profile and the reliability of each component. It allows to find existent bottlenecks that are affecting negatively the overall reliability, such as components that are overused or connectors that need to redistribute the system load. Our approach differs from these by applying a non-linear variation on component reliabilities, addressing the issue of exceeding the range of possible values. In addition, our work is applied directly from an ADL specification, making it possible to automatically build a suitable stochastic model to perform reliability prediction, discarding the issues carried by manual activities (time-consuming and prone to errors).

Our work tries to address some shortcomings identified by research surveys [Barais *et al.*, 2008; Goseva-Popstojanova *et al.*, 2005; Immonen & Niemelä, 2008], such as the lack of automated processes for reliability analysis and poor method validation. Thus, in this chapter we address these shortcomings by encompassing a thorough analysis on reliability that allows to evolve a system and, at the same time, providing support for the architect on paramount decisions and insightful architectural trade-offs.

4.7 Summary

A sensitivity analysis of a specific non-functional property determines which architectural elements require attention to improve the quality of the system. For this reason, our approach varies the reliability of the components and usage profile to allow architects to examine about the system quality, identify key issues and support architectural evolution.

Hence, our method supports an analysis of the reliability and usage profile to find its sensitive architectural constituents. This was achieved by employing a novel technique to vary reliability and a ranking system that orders components with respect to their influence on the overall system reliability. Moreover, we demonstrated the applicability of our approach by analyzing the reliability of the architecture provided. The information gathered in the analysis resulted in an improved system which reduced the unreliability by one third.

Finally, we automated the analysis procedure by varying the architectural parameters in the stochastic model. In this way, our approach provides the means for performing a thorough analysis without requiring any effort or specialized knowledge on the part of the architects to improve or evolve the designed artifact. We believe this study has the potential to encourage the widespread adoption of what-if simulations by practitioners without the complexity and error-inducing potential of manual approaches.

Chapter 5

Implementation and Validation

Automated methods to estimate reliability early in the software development are a great asset to architects since they enable them to test and analyze their own products. These methods, allow them to improve their own designs without the need to manually construct stochastic models and reconstruct them whenever a change is made to the architecture. With this in mind, we implemented a tool – Affidavit (5.1) – for the assessment of reliability which has analysis capabilities accessible from current design tools and frameworks. Our approach was validated by comparing our results with those of other approaches. In particular, we apply the same scenarios and failure data as other studies (5.2) to validate our reliability prediction method with and without reference to architectural styles (5.3).

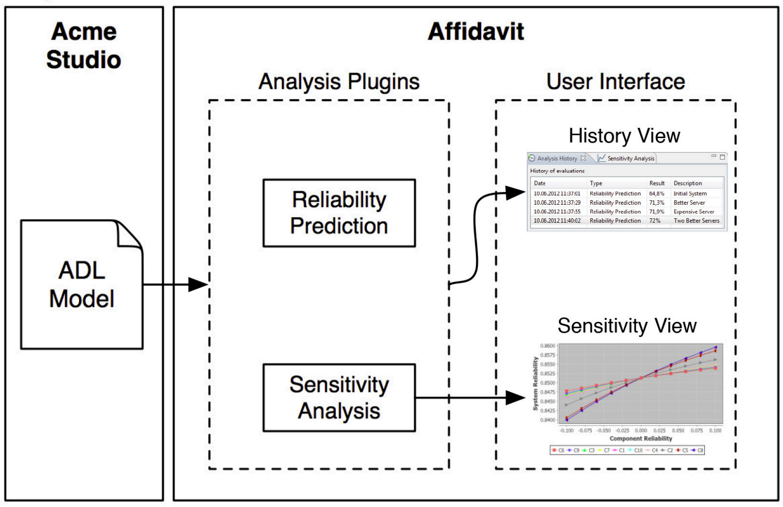
5.1 The Affidavit Tool

The assessment of quality attributes in an early phase of the development serves as a guidance for architects to reflect on software design as well as to identify key issues. However, current practice lacks on processes, methods or techniques to automatically estimate several quality attributes, namely reliability.

To overcome these limitations, we propose Affidavit as a tool that automatically makes assessments and performs a thorough analysis of the reliability of a software architecture. This is described using the Acme Architectural Description Language (ADL) [Garlan *et al.*, 1997] and was developed as a plugin for AcmeStudio [Schmerl & Garlan, 2004], designed to provide automated reliability prediction [Franco *et al.*, 2012] and analysis [Franco *et al.*, 2013] within a graphical Integrated Development Environment (IDE).

5.1.1 Implementation

The aim of Affidavit is to offer testing and analysis capabilities within design tools and frameworks used by architects. In short, this work intends to be closer to architects and ready for use by a simple plugin installation. Affidavit is available as a plugin in the AcmeStudio framework and is depicted in the diagram in Figure 5.1.



Date	Type	Result	Description
10.06.2012 11:37:01	Reliability Prediction	64,8%	Initial System
10.06.2012 11:37:29	Reliability Prediction	71,3%	Better Server
10.06.2012 11:37:55	Reliability Prediction	71,9%	Expensive Server
10.06.2012 11:40:02	Reliability Prediction	72%	Two Better Servers

Figure 5.1: Affidavit diagram

The diagram of Affidavit depicts the use of the Acme Architectural Description Language (ADL) as input to generate a stochastic model that displays the failure behavior of the system. This is possible with the aid of the Acme Tool Developer's Library (AcmeLib)[Garlan *et al.*, 1997] developed by Software Engineering Institute (SEI) at the Carnegie Mellon University (CMU). This library allows third-party applications to parse the content of a Acme file and manage the architectural model.

Thus, our approach provides two alternatives for the architect: (i) predict the reliability or (ii) perform an analysis of the architecture under design.

The former implements the method outlined in Section 3.7 in which we translate the ADL into a Discrete-Time Markov Chain (DTMC). The resulting DTMC is solved by the Prism Model Checker Tool that provides a quantitative reliability prediction of the architecture under design. The result allows architects to investigate the proposed design and make changes and comparisons as well as applying different styles to improve the predicted system reliability.

The latter applies the method discussed in Section 4.5 by performing a reliability or usage profile analysis on the designed architecture. Our approach automatically generates a DTMC for each variation performed in the system and ranks components and interconnections, in terms of their impact on the system reliability.

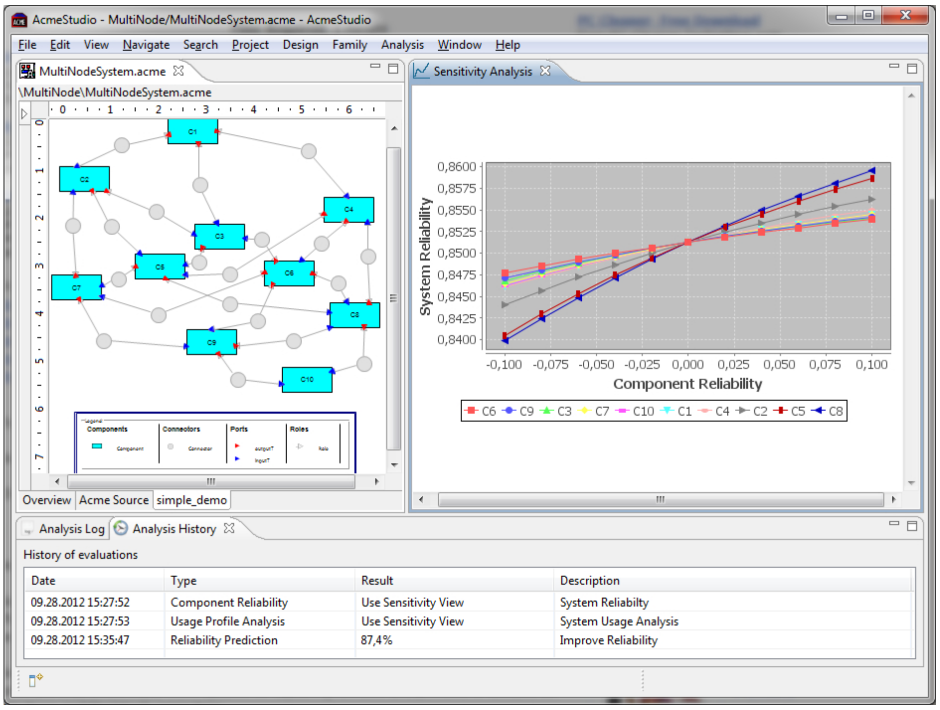
The results of both the prediction and analysis can be observed in the "History View" depicted in Figure 5.1. This view serves as a log of the modifications performed to the system and provides a description of the experiment along with its 'reliability outcome'. This view enables architects to find a previous arrangement and compare architectural alternatives through the result of their reliability. When a sensitivity analysis is performed, Affidavit shows a graphical representation of the obtained results, illustrated by the "Sensitivity View".

The next section gives an example of the information provided in the Affidavit Graphical User Interface (GUI).

5.1.2 GUI Example

The procedure of analyzing an architecture by the Affidavit tool is exemplified in Figure 5.2.

As can be seen, the "History View" depicts the tasks carried out for the designed architecture: reliability prediction or sensitivity analysis. Reliability prediction is shown as a percentage value and is useful for comparing different architectures, and giving user guidance on what are the differences between one architecture and another. As regards the sensitivity analysis, this is shown through a graph that relates the variations performed (x-axis) with the overall system reliability (y-axis) where each line corresponds to a component. This type of analysis assists the architects in deciding which components should be improved, by identifying their impact on the overall quality of the system.


AcmeStudio - MultiNode/MultiNodeSystem.acme - AcmeStudio
File Edit View Navigate Search Project Design Family Analysis Window Help
MultiNodeSystem.acme
\MultiNode\MultiNodeSystem.acme
Sensitivity Analysis
System Reliability
Component Reliability
C6 C9 C3 C7 C10 C1 C4 C2 C5 C8
Overview Acme Source simple_demo
Analysis Log Analysis History

History of evaluations

Date	Type	Result	Description
09.28.2012 15:27:52	Component Reliability	Use Sensitivity View	System Reliabilty
09.28.2012 15:27:53	Usage Profile Analysis	Use Sensitivity View	System Usage Analysis
09.28.2012 15:35:47	Reliability Prediction	87,4%	Improve Reliability

Figure 5.2: Overview of the Affidavit Tool

5.1.3 Experiments

The main goal of Affidavit is to assess and analyze system reliability from an architectural perspective. We modeled two different scenarios to demonstrate the usefulness and validity of our tool. The used architectures and reliability values specified in this demonstration were not extracted from a real system and do not represent a real scenario. Their purpose is only to support our method and show the applicability of our tool. The two used scenarios have the same architectural elements and reliability values, but they differ in the arrangement of the components and in the applied architectural styles. A reliability value output is obtained for each scenario from this arrangement and we conducted a detailed analysis to establish which components or connections are affecting the overall system reliability.

5.1.3.1 Scenario #1

The system used in this scenario, illustrated in Figure 5.3, is composed by a fault-tolerant architecture with two equivalent systems. Each fault-tolerant sys-

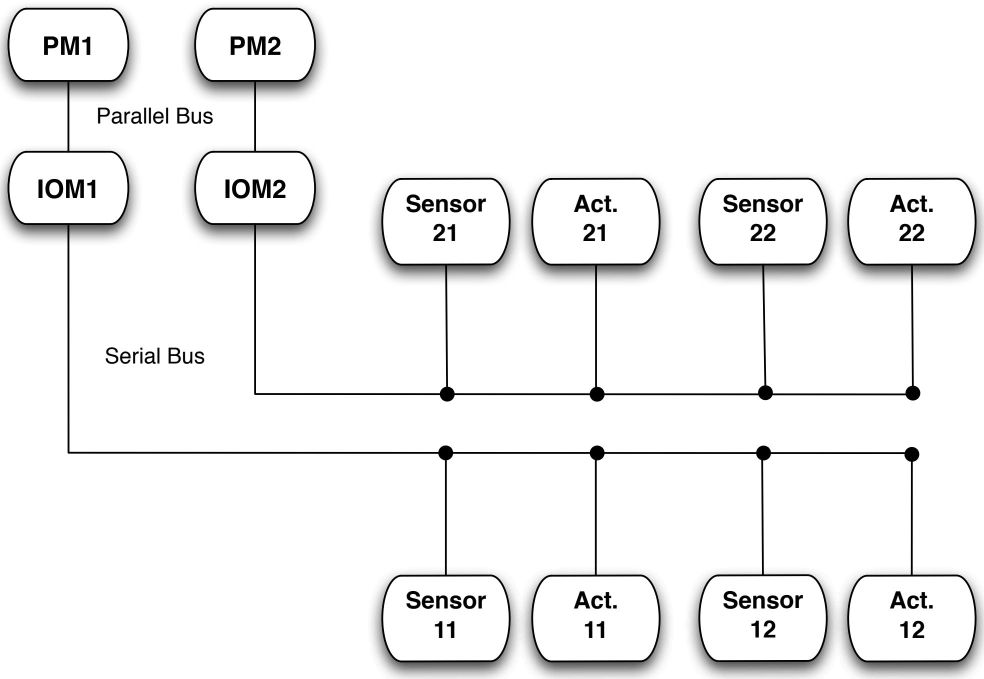


Figure 5.3: Scenario #1 – Diagram

tem entails a reactive system that monitors the environment through sensors, processes the information and acts with a predefined plan. In detail, the system embodies the following elements:

- A Processing Module (PM);
- A Parallel Bus (PB);
- Serial Bus (SB);
- An Input/Output Module (IOM);
- Two Sensors (Sensor 1 and 2);
- Two Actuators (Act. 1 and 2).

Each sensor monitors a different function in the system and will invoke its own actuator (*i.e.*, Sensor1 invokes Actuator1, but not Actuator2). With regard to the total number of requests performed by the system, we specify the usage profile as follows: 40% of the requests are resolved by Sensor1 and the other 60% are resolved by Sensor2.

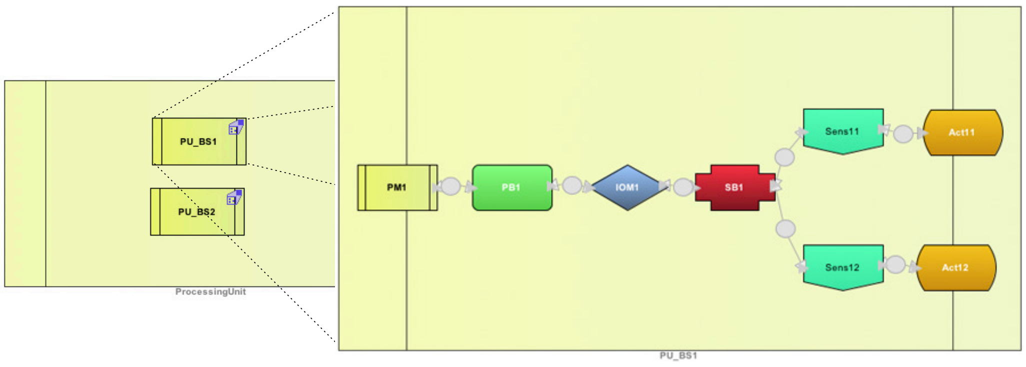


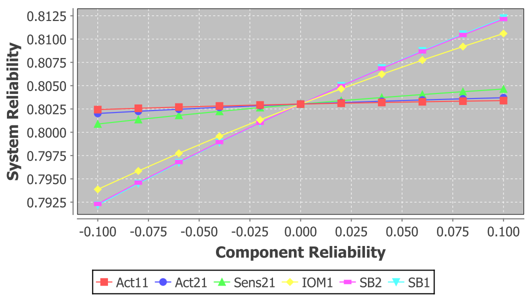
Figure 5.4: Scenario #1 – Software Architecture

Thus, we modeled this scenario in AcmeStudio, as illustrated by Figure 5.4, and specified the reliability values in accordance with what is shown in Table 5.1.

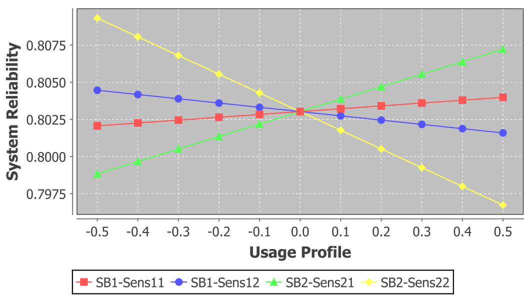
Table 5.1: Scenario #1 – Component reliabilities

Component	PM1	PB1	IOM1	SB1	Sens11	Sens12	Act11	Act12
Reliability	0.95	0.98	0.85	0.80	0.99	0.92	0.99	0.95
Component	PM2	PB2	IOM2	SB2	Sens21	Sens22	Act21	Act22
Reliability	0.95	0.98	0.85	0.80	0.94	0.91	0.98	0.93

Affidavit performed an analysis in the system that determined that the reliability of the modeled system is of 80.3%. This means that from the total number of requests performed to the system, 19.7% fail for some reason and cannot be successfully resolved.



(a) Variation of the reliability of components



(b) Variation in the system usage profile

Figure 5.5: Sensitivity Analysis of Scenario #1

A thorough sensitivity analysis to identify architectural bottlenecks was conducted in accordance with the method outlined in Chapter 4 and the obtained results are given in Figure 5.5. In detail, Graph 5.5(a) depicts the variation of 10% of the reliabilities of the different components in the system. In this graph only the three best and worst reliability variations are illustrated from a total of sixteen components. We rank the variations by calculating the derivative of the reliability around the point where the variation is null (*i.e.*, variation of 0%). This ranking is shown through the graphic caption, where, from the left to the right, the components are ordered in terms of a lower to a higher increase of the impact on the overall system reliability.

In this scenario, components Act11, Act21 and Sens21 are those in which their variation has less impact on the overall reliability, and can be considered to be the diminishing returns in the system. On the other hand, components SB1, SB2 and IOM1 are those where their variation has the highest impact.

Regarding usage profile, Graphic 5.5(b) shows a variation of 50% in the load of requests that are performed from the SB component to the Sensors. The connection from SB1 to Sens11 shows that it is an already diminishing return, but the increase in the usage profile from SB2 to Sens21 leads to an improvement in the overall system reliability. In addition, Graphic 5.5(b) informs architects that the load in the connection SB2 to Sens22 and Sens21 is not suitably balanced and should be subject to change to obtain the maximum benefit from the system reliability.

5.1.3.2 Scenario #2

In this scenario we applied the same architectural elements, reliability values and usage profile as in Scenario #1. However, as can be seen from Diagram 5.6, it differs in the applied styles and in the architectural structure. Specifically, we

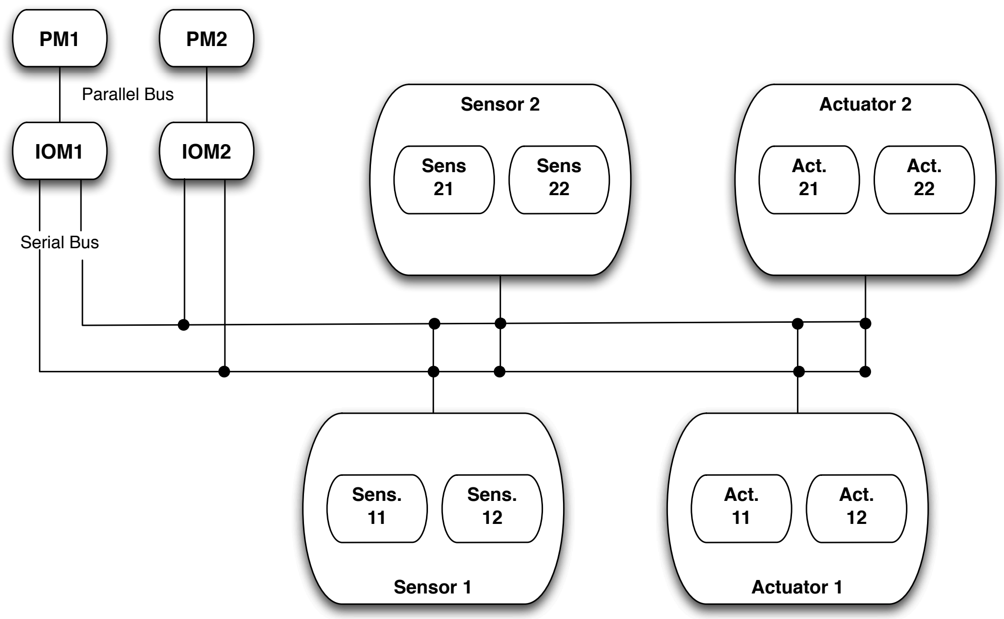


Figure 5.6: Scenario #2 – Diagram

put together the two Serial Buses in parallel so that they could act as a backup for each other, this acts as fault-tolerant communication channel. In addition, we joined the Sensors and Actuators from the previous scenario and we setup them in a fault-tolerant redundant arrangement.

Figure 5.7 illustrates the system representation in AcmeStudio, which was used for predicting and analyzing system reliability.

In this scenario the predicted reliability was 91.1%, which represents an increase of 10.8% to the previous scenario. It must be stressed that the components are the same, but have simply been arranged differently.

Figure 5.8 depicts the sensitivity analysis performed on the system.

The variation in the reliability of the system components is illustrated in Graph 5.8(a) and since we used the same reliability values as in Scenario#2, the results are identical. The only difference is that the component Sens21 in

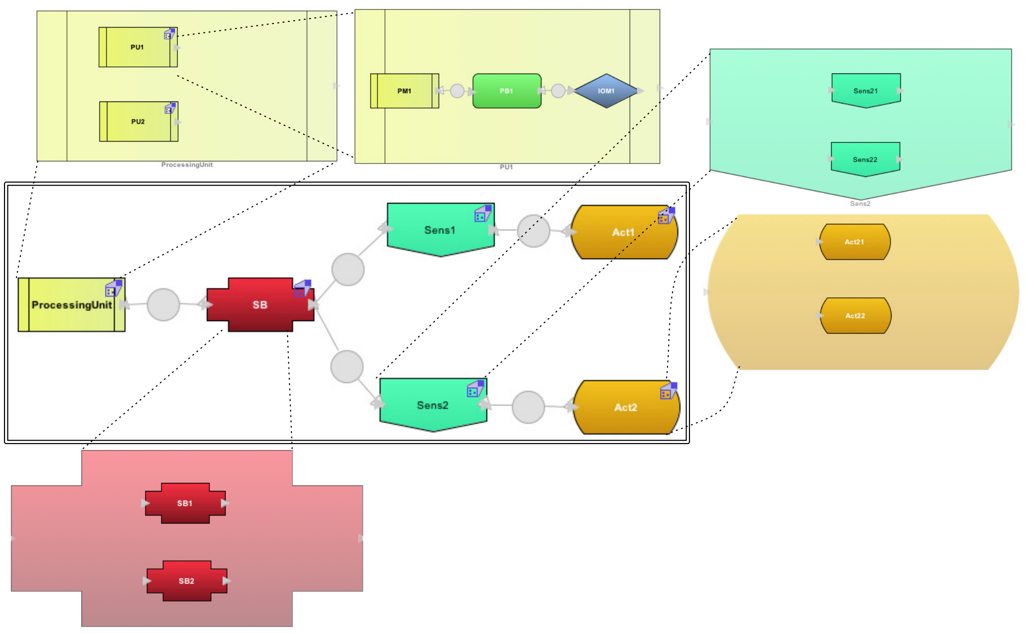
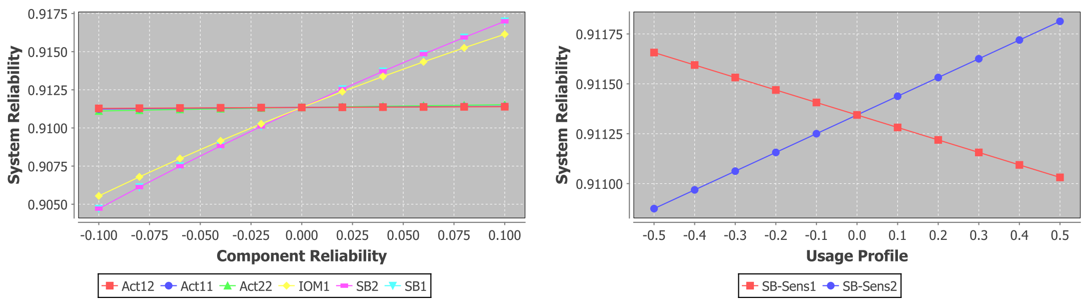


Figure 5.7: Scenario #2 – Software Architecture



(a) Variation in the reliability of components

(b) Variation in system usage profile

Figure 5.8: Sensitivity Analysis of Scenario #2

the previous scenario is replaced by Act22 as one of the diminishing returns. With regard to the usage profile, Graph 5.8(b) shows that increasing the load of requests on the connection SB-Sens2 would increase the system reliability.

5.1.3.3 Conclusion

The examples provided show that the Affidavit tool is able to assess different architectures while taking account of the distinct styles applied. In addition, it provides the means for architects to compare, test and validate different architectural alternatives that would fulfill the quality requirements established by the stakeholder. Finally, the sensitivity analysis recommends architects about what should be the future direction to improve the system.

As an example of the valuable information obtained from a thorough analysis, in Scenario#2 we increased by 10% the reliability of components SB1, SB2 and IOM1, as well as, increasing the load of requests sent to the component Sens2 by 20%. this resulted in an increased reliability of 92.7% which represents an improvement of 12.4% and 1.6%, when compared with Scenario #1 and #2, respectively.

5.2 Automated Reliability Prediction Validation

To validate the accuracy of our reliability prediction method, we compared the results of our approach with the values obtained from the studies of Cheung [1980], Lo *et al.* [2005] and Gokhale & Trivedi [2002].

All the compared publications use the same architecture, as illustrated in Figure 3.1(b), and assign a set of different reliability values for components, as depicted in Table 5.2.

The same software architecture, usage profile and reliability values were applied from each of the publications to compare and validate our results. With regard to other approaches, Cheung [1980] used a composite method through an absorbing DTMC to predict the reliability of an architecture. Lo *et al.* [2005] made use of a hierarchical method to predict the reliability and Gokhale & Trivedi [2002] showed the results obtained from using both methods of the state-based approach, the composite and hierarchical methods.

Table 5.3 shows the obtained reliability results between the different approaches.

Table 5.2: Component reliabilities

R_i	**Cheung [1980]**	**Gokhale & Trivedi [2002]**	**Lo *et al.* [2005]**
1	0.999	0.999	0.99
2	0.980	0.980	0.98
3	0.990	0.990	0.99
4	0.970	0.970	0.96
5	0.950	0.950	0.98
6	0.995	0.995	0.95
7	0.985	0.985	0.98
8	0.950	0.950	0.96
9	0.975	0.975	0.97
10	0.985	0.985	0.99

Table 5.3: Validation of the reliability prediction method

		Literature	**Our Approach**	**Difference**
Cheung [1980]		0.8299	0.8299	0.0
Lo *et al.* [2005]		0.8482	0.8512	0.003
Gokhale [2002]	Composite	0.8299	0.8299	0.00
	Hierarchical	0.8280		0.0019

Viewed in detail, our method provides similar results when both composite and hierarchical methods are compared. In the former, our method has no difference, and in the latter - hierarchical methods - it has a maximum of 0.003 difference. It can be concluded from these results that our approach shows identical values which validate the correctness of our reliability prediction method.

5.3 Validating Architectural Styles

In the second part of our validation procedure, we confirmed that our approach generates a correct mathematical model and provides an accurate reliability value when an architectural style is applied.

The input architectures used to test the validity of our approach are those shown in Section 3.6. They were modeled on Acme ADL and our approach was adopted to generate the mathematical model. Finally, they were loaded into the Prism model checker tool, to check and simulate the architecture. We tested the fault-tolerant style with one active and two redundant components and the parallel style with three parallel components.

The comparison between the results obtained from our approach and the ones achieved through the methods employed by Wang *et al.* [2006], are depicted in Table 5.4.

Table 5.4: Validation of the architectural styles

Style	**Wang *et al.* [2006]**	**Our Approach**	**Difference**
Batch-Sequential	0.9248722	0.9248722	0.00
Parallel	0.8945088	0.8945088	0.00
Fault-tolerance	0.9503923	0.9503923	0.00
Call-and-return	0.9317644	0.9317631	≈ 0.00

With regard to the values provided by previous research studies, our results are identical, which shows that our approach generates accurate and correct mathematical models when using architectural styles.

5.4 Summary

As a means of supporting the design phase of the software development, we implemented a plugin that could be integrated in a current architectural framework tool. Our goal is to provide the actual means by which practitioning architects can test and analyze their products with a minimum effort. The provided plugin performs a reliability assessment and a thorough analysis of the architecture to find weak points in the architecture that could be improved.

To test the validity of our reliability prediction method, we applied the same case-studies used by other research approaches to compare the obtained results. The comparison process shows that when employing the composite method, our automated approach had exactly the same result and differed at most by 0.003% from the hierarchical one. Moreover, we tested our approach when dealing with different architectural styles and the results showed identical reliability outcomes. In light of the obtained data, it can be concluded that our approach has validity when compared with classical and state-of-the-art reliability methods.

Chapter 6

Application to Self-Adaptive Systems

In the modern world, systems are becoming more autonomous and independent in an attempt to relieve humans of having to carry out routine actions such as driving or doing housework, by introducing self-driving cars, Roomba^{TM} vacuum cleaners or automated cooking robots. These self-ruling autonomous machines are today's future, some inspired by sci-fi movies of Hollywood, others originating from the mind of creative people. However, these systems share a common-base in which they are able to modify their structure or behavior during runtime in order to meet specified goals, such as a clean house or driving from point A to B.

These systems are monitored to obtain runtime properties which are analyzed to identify conditions where the system may be deviating from the desired quality goals. In these situations, adaptation courses are planned and executed to get the system into the right track. However, in view of the critical nature of the system, adaptations should be planned with care and take account of every possible scenario. For example, self-driving cars have strict safety requirements and whenever an adaptation is planned, it must ensure that no human lives (passengers or pedestrians) are put at risk.

The area of self-adaptive systems addresses the issue of autonomous software and hardware through sensory feedback mechanisms. Current adaptation approaches vary from simple algorithms that are condition-action based to other, more complex approaches that involve Markov Decision Processes (MDPs) or utility-theory [Cheng, 2008]. Generally, these decision-making algorithms act according to a predefined set of operators considered to be static [Fredericks *et al.*, 2013; Macías-Escrivá *et al.*, 2013]. These static operators defined by humans are only effective in the specific domains or expected contexts in which they have been configured [Salehie & Tahvildari, 2009]. In systems with a high number of runtime and environment variables, the possible adaptation scenarios and consequences may rise exponentially, and become almost unfeasible for humans to reflect on in every possible combination. As a result, under unexpected conditions a system may fail to select the best strategy which may lead to a degradation of the provided services.

To overcome the limitation that occurs with static operators responsible for triggering adaptations, we propose a method that automatically predicts whether an adaptation strategy fulfills the desired non-functional goals, even in unexpected conditions. This involves adopting the approach outlined in Chapter 3 to generate stochastic models at runtime. In this way, we are able to generate a model for each strategy to evaluate their impact on the failure behavior of the system. The result allows the system to select a proper adaptation that has a positive impact on the desired quality attributes.

In short, our proposal seeks to improve the planning phase of self-adaptive systems, by automatically anticipating the effect of each adaptation impact on non-functional properties. In view of this, this chapter sketches the background of self-adaptive systems (6.1) and examines how the automated generation of stochastic models can be integrated with those systems (6.2). Thus, an experiment (6.3) was carried out to validate the effectiveness of our approach in predicting the outcome of an adaptation strategy, and finally, we examine the obtained results (6.4). Finally, we describe the related work to our approach (6.5) before summarizing the contributions of the proposed method (6.6).

6.1 Self-Adaptive Systems

The interest in self-adaptation by the research community can be noted by the number of published research articles. In particular, it is worth recording that the number of articles with the term *self-adaptive* in its title was 6.690 in 2008 and in 2012 this number rose to 10.800, an average increase of 1.027 submitted articles every year[1]. With the rise in interest in this topic, the number of proposed self-adaptive solutions also increased.

In short, a system is considered to be self-adaptive when it modifies its own behavior in response to changes in its operating environment [Oreizy *et al.*, 1999]. A widely adopted self-adaptive approach is the MAPE-K model developed by the IBM Autonomic Computing Initiative [IBM Corp., 2004]. Its conceptual model is depicted in Figure 6.1, and entails a separation of the adaptive phases. In specific

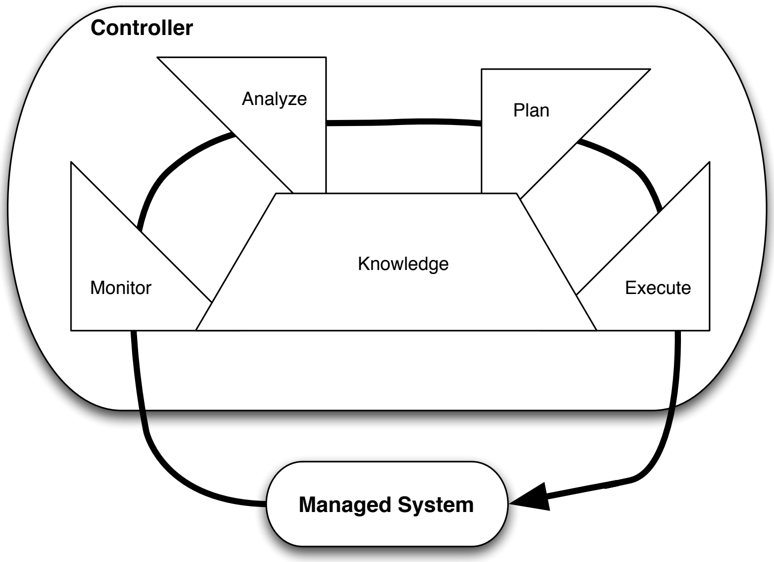


Figure 6.1: The IBM Autonomic MAPE reference model

terms, this model illustrates a closed-loop process, and distinguishes between the controller, an entity responsible for handling the adaptation process, and the managed system, the entity subject to adaptation (*i.e.*, target system). The MAPE-K includes a knowledge-base which abstracts the system, contains data, and models decisions and behavior, by enabling the separation of adaptation responsibilities and allowing the different MAPE phases to be coordinated. It has the following functions:

- Monitoring – it collects data about relevant properties from the managed system through probes;

[1]Values extracted from the Google Scholar engine.

- Analyzing – it detects if the system is in a condition to be adapted. This occurs when a property value is outside an expected range or quality goals become degraded;
- Planning – this determines a course of action based on quality goals, different strategies and a system model;
- Execution – this carries out the course of action planned in the previous phase to adapt to its behavior.

6.2 Reliability Prediction within Self-Adaptation

To solve the problem of planning an adaptation through static operators we propose a quantitative prediction method to ensure the desired goals are achieved. We focus on failure avoidance goals and as such, we propose employing a method to predict the failure behavior of a system. In detail, the proposed method from Chapter 3 is employed to support the planning phase of a self-adaptive system. Figure 7.2 provides an overview of our approach and its integration into the different phases of the MAPE-K loop.

In concrete terms, our approach begins by collecting runtime metrics from a running system to update its architectural description. This process ensures that at each recurring analysis phase, the model of the system is updated to the current environment and system conditions. In the *Analyze* phase, our approach makes a copy of the software architecture by following each possible adaptation strategy and applies its changing operators. These operators are defined as the changes that each strategy would perform in the managed system if they were selected. Thus, our method generates a model that represents the system behavior for each adaptation strategy. To solve the generated model, our approach relies on a model checking tool to predict the quality of the impact. In the final stage, our approach supports the planning phase by updating constant weights or impact vectors which can be used for comparative purposes and helps decide what is the best strategy to achieve the desire quality goals.

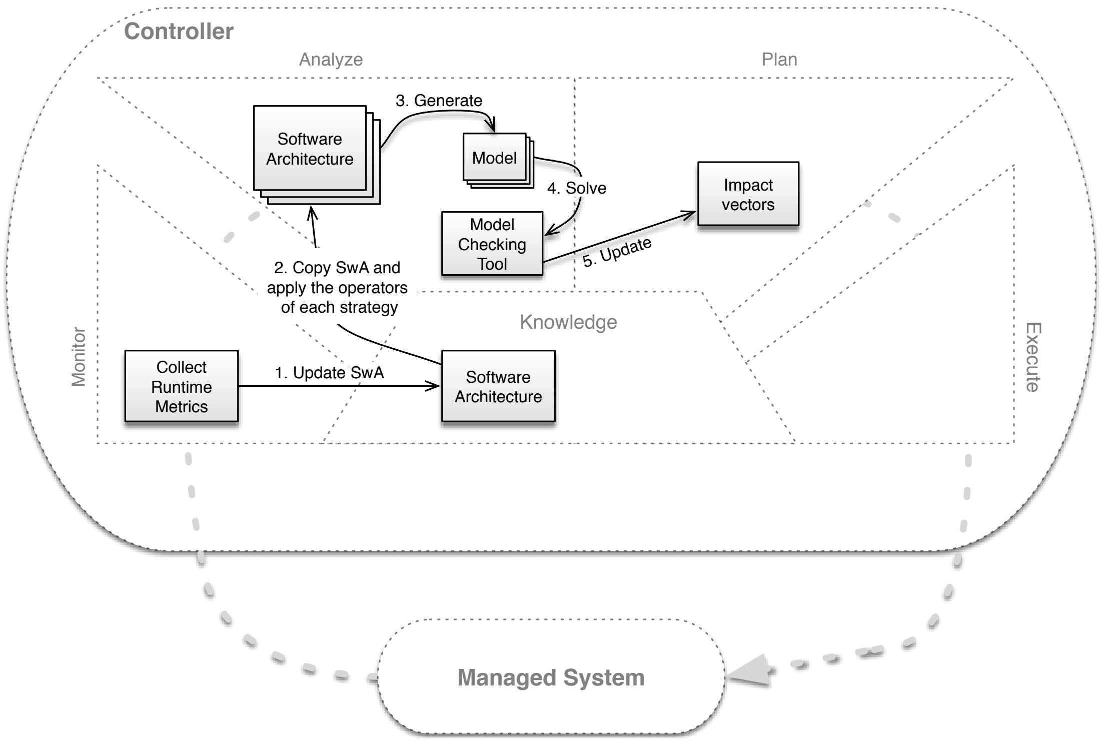


Figure 6.2: A general overview

6.3 Case-Study

To test the effectiveness of our approach, we adopted a "de facto" standard case-study from the self-adaptive community, the Znn.com [Cheng, 2008]. In this section, we outline the experimental setup along with an example of our approach by generating stochastic models for each adaptation strategy.

6.3.1 Adopted self-adaptive system

In response to the recent rise of interest in the self-adaptive topic, Villegas *et al.* in 2011 established a framework to evaluate self-adaptive solutions by classifying them through a set of dimensions and introducing a set of metrics to assess how adaptations are performed. Examples of the metrics used to evaluate different adaptive systems include the following: accuracy (*i.e.*, whether adaptation goals are met), overshoot (*i.e.*, the amount of computational resources used) or even, settling-time (*i.e.*, the time the system needs after an adaptation to achieve the desired state).

On the basis of the study carried out by Villegas *et al.* [2011], we chose the Rainbow self-adaptive system [Cheng & Garlan, 2012; Cheng *et al.*, 2009; Garlan *et al.*, 2004] which supports quality-driven goals and quantitative metrics. Moreover, the selected solution is based on the MAPE-K approach from IBM Autonomic Computing Initiative and makes use of the Acme as the basis Architecture Description Language (ADL) as does our proposed method which is outlined in Chapter 3. Rainbow is an architecture-based self-adaptive system designed by the Carnegie Mellon University and its framework is depicted in Figure 6.3 which shows that it consists of two main subsystems: the controller and target.

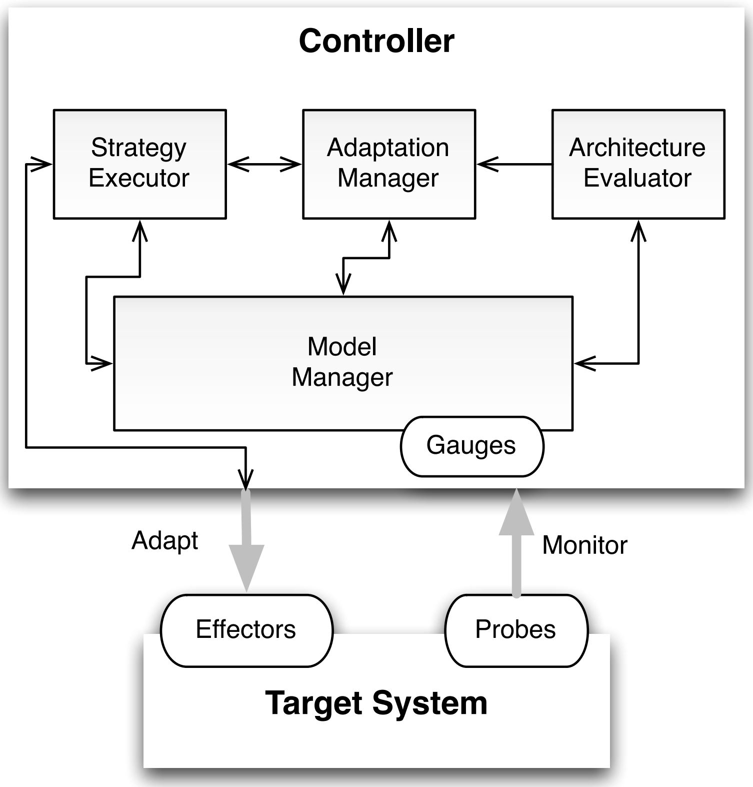


Figure 6.3: The Rainbow framework

The controller monitors the target system through probes and gauges which update properties in the architectural model managed by the Model Manager.

The Architecture Evaluator evaluates the model to determine if the system is operating within an acceptable range of quality goals. If the evaluation finds that the system is not operating under normal conditions, it invokes the Adaptation Manager which is responsible for selecting a more suitable adaptation strategy. Each strategy involves a bundle of simple courses of action denoted as adaptation tactics. After a strategy has been selected, the Strategy Executor is responsible for applying a sequence of actions to the target system, so that the selected strategy is instantiated in the system.

The target system is defined as the resource that will be monitored and adapted to meet the self-adaptation goals. The environment consists of the external world that interacts with the target system. It is considered to be non-controllable and at the same time, capable of influencing the runtime properties (*e.g.*, hardware, physical context or network).

6.3.2 Target system

We defined the target system as being the Znn.com, a typical infrastructure for a news website and its diagram is depicted in Figure 6.4. It has a tiered architecture

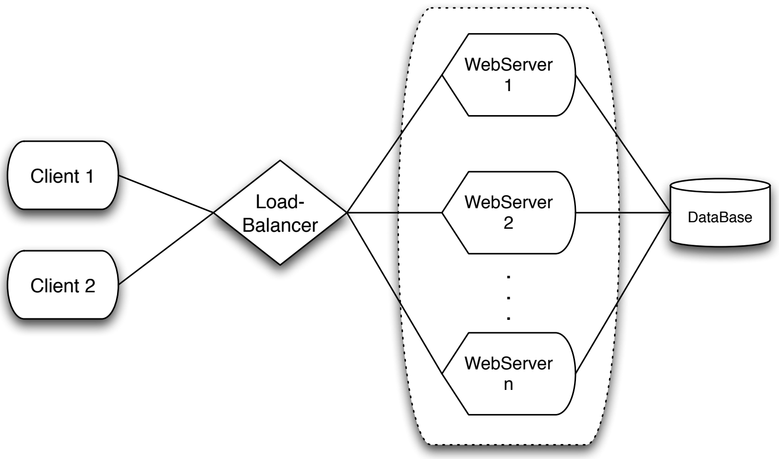


Figure 6.4: The Znn.com diagram

with a set of web-servers that serve content, both textual and graphical, from back-end databases to clients through a front-end presentation logic. In addition, it uses a load-balancer to reroute the requests from the client to a pool of servers.

The number of active servers will depend on the selected adaptations required to fulfill the system goals. It must be stressed that Znn.com is an actual platform running in actual servers using a standard Apache distribution. Znn.com is not a simulation model.

6.3.3 Experimental design

Self-adaptive systems are designed with a set of operators, weights, preferences or values to support their decision-making and drive adaptations to meet desired quality goals. In this section, we define the static operators that are usually configured by the human operator in the Znn-like system, and that must be updated automatically by the self-adaptive middleware in accordance with the quality predictions provided by our approach.

6.3.3.1 Adaptation goals

As with a typical news provider, Znn.com focuses on providing news content to its customers in a reliable way while keeping the operating costs to a minimum. In short, we identify three quality objectives for self-adaptation:

- Availability – this expresses the probability that the system is operating properly when it is requested for use. In specific terms, it is a long-run measure that takes into consideration the reliability of each time-frame and the 'repair actions' as adaptations executed in the target system. The goal of this quality attribute is to maximize its potential, even if it has to incur a higher operational cost;

- Operational Cost – this measures the number of computational resources that need to be available during the experiment. Each server in the pool of servers is deployed through a Virtual Machine (VM) and the goal is to reduce the number of VMs that are being kept to a minimum. For example, the system switches off virtual machines when processing a low number of requests;

- Utilization – this defines the amount of work received by the system in terms of the maximum load that is supported by all the available servers. For example, if the current work that is being processed reaches the maximum capacity that all the servers are able to process, the system may have to adapt by enlisting a new server to increase the total load that can be handled.

6.3.3.2 Adaptation metrics

We monitored the target system to collect data for each ten second time-frame — the period between adaptations — and obtained the following runtime properties:

- Active resources: the number of webservers that are active and responding to requests;

- Reliability: defined as *"functioning correctly"* [Storey, 1996] and in this case-study it refers to the number of non-failed requests between recovery actions (*i.e.*, adaptations). Hence, we define the failure behavior as a request that takes an unreasonable time to receive a response (*i.e.*, $> 2000\ ms$) or when the returning code is not successful (*i.e.*, HTTP status code $\neq 200$);

- Load: indicates the number of requests that have been responded to within the time-frame over the total maximum capacity the system can hold.

6.3.3.3 Adaptation strategies

The selected adaptations of this particular case-study focus on the server pool. Hence, depending on the current state of the runtime properties of the system, the controller may select one of the following strategies:

- Enlist server – Enables a server, if there is a spare one ready to be activated;

- Discharge the slowest server – If there are at least two active servers and no failure has occurred, our approach will discharge the slowest one (*i.e.*, the one with the highest value of mean response time);

- Discharge the least reliable server – If there are at least two active servers and at least one failure has occurred, the system will discharge the less reliable (*i.e.*, the server with the highest failure rate).

6.3.3.4 Static adaptation operators

The self-adaptive solution used in this study, Rainbow, uses utility-theory as its decision-making algorithm. This type of algorithm relies on utility functions, preferences and impact vectors to ensure that adaptations fulfill the defined quality goals which are listed below.

Utility functions The utility-theory measures the monitored system properties according to a utility function. It provides a score which reflects how properties are behaving when seeking to achieve the proposed goals. These values and functions are defined in the design phase of the self-adaptive system. An illustrative example is given in Table 6.1. The values that fall in intermediate points are linearly extrapolated. From Table 6.1 it can be inferred that the utility is higher for reliability values close to 100% and lower values are heavily punished because of their utility function. In addition, the defined values lead to a low consumption of computational resources by designating one active web-server as the best utility outcome of the system. With regard to the experienced load, the system favors a low utilization capacity which makes it possible to decide, when the number of requests increases, and whether it is better to activate one more active resource or allow the load to increase.

Utility preferences The preferences define the relative importance of the quality dimensions and serve as an example to prioritize quality attributes. They are shown in Table 6.2 and it can be noticed that availability is twice as important as the cost or utilization of the system.

Table 6.1: Utility Functions

Reliability		Active Resources		Load	
Value (%)	Utility	Value	Utility	Value (%)	Utility
100	1.0	0	0.0	100	0.0
99	0.88	1	1.0	90	0.05
95	0.54	2	0.85	80	0.1
90	0.3	3	0.55	70	0.4
85	0.16	4	0.3	60	0.5
80	0.09			50	0.6
75	0.05			40	0.7
50	0.002			30	0.8
0	0			20	0.9
				10	0.95

Table 6.2: Utility preferences

	Percentage
Availability	50%
Cost	25%
Utilization	25%
Total	100%

Although they may not be achieved optimally, these preferences can be used to solve resource constraints (*e.g.*, on a server discharge, the remaining servers may not be able to process the current demand of requests) or trade-offs between certain quality attributes.

Impact Vectors The impact of a strategy on each of the quality dimensions is represented as a vector of cost-benefit values between the strategy and each quality dimension. Table 6.3 shows the adopted values in our case-study and it should be noted that our *Enlist Server* strategy increases both availability and the used computational resources, at the same time that it reduces the system utilization, since it has more machines to process the same demand of requests. Conversely, both *Discharging Server* strategies will reduce costs and increase the use of the system. When deciding to discharge an unreliable server, the system will increase availability, but it will be kept the same when discharging the slowest server from the pool.

Table 6.3: Static impacts on the quality dimensions for each strategy

	Availability	Cost	Utilization
Enlist a Server	+10%	+1.0	-20%
Discharge Unreliable Server	+1%	-1.0	+20%
Discharge Slowest Server	0.0%	-1.0	+20%

Utility rate calculations Adaptation strategies are ranked through the utility metrics shown above and illustrated in the following formula:

$$\begin{aligned}\text{Utility} =&(\text{reliability } + \text{Impact } (\textit{e.g.}, +10\%)) \times \text{Availability Preference } (50\%)\\ &+ (\text{active resources } + \text{Impact } (\textit{e.g.}, +1\%)) \times \text{Cost Preference } (25\%)\\ &+ (\text{load } + \text{Impact } (\textit{e.g.}, \text{-}20\%)) \times \text{Utilization Preference } (25\%)\end{aligned}$$

To calculate the utility for each strategy we rank the current monitored properties together with their impact on the quality goals when applying the proposed adaptation strategy. The controller performs this assessment to plan an adaptation that best achieves the non-functional requirements of the system. As a result, the controller chooses the strategy that has the highest utility value.

6.3.4 Example

To show the effectiveness of applying quantitative verification methods at runtime, we provide a demonstrative example that provides details of the generation of the stochastic model and utility calculations. Figure 6.5 illustrates the architectural model of the Znn.com taken from a snapshot of an actual run of the system.

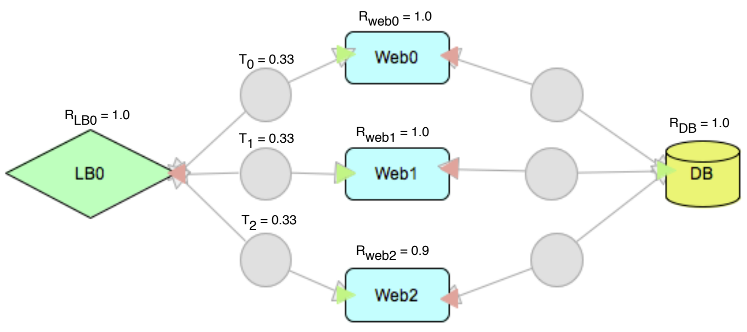


Figure 6.5: Demonstration of the Architectural Model

The runtime metrics obtained from this time-frame are as follows:

- Reliability: 95.7%;
- Active Resources: 3 active web-servers;

- Load: 40%.

This demonstration shows three active web-servers one of which is failing to respond to client requests (*i.e.*, Web2). In these circumstances, only three adaptation strategies are eligible: non-adaptation, enlist server and 'discharge the least reliable'. The strategy of 'discharging the slowest server' is not eligible, because failures have occurred in the system (explained in Section 6.3.3.3). To cope with the failure and conduct the system to achieve the desired goals, an adaptation strategy must be selected and executed to alter the current behavior. An adaptation plan is determined through assessing the utility outcome from the possible strategies and selecting one that has the best utility outcome. In the following sub-sections, we outline the calculation process of the two scenarios under assessment: constant weights and their dynamic update. The former entails adopting traditional self-adaptive approaches with constant weights and static adaptation operators. The latter describes our approach by predicting the behavior of each strategy and dynamically updating these static operators.

6.3.4.1 Constant weights

This test shows the utility calculations for the traditional self-adaptive approach which rely on constant weights or impact vectors. In this particular time-frame, the static approach will pick the strategy with the highest utility, the Enlist Server, as will be demonstrated.

No adaptation

$$\begin{aligned}\text{Utility}_{\text{NA}} &= 0.6\ (95.7\%\ \text{reliability}) \times 50\% + \\ &+\ 0.55\ (3\ \text{active resources}) \times 25\% + \\ &+\ 0.7\ (40\%\ \text{load}) \times 25\% \\ &= 0.6125\end{aligned}$$

Enlist Server

$$\begin{aligned}\text{Utility}_{\text{ES}} &= 1.0\ (95.7\% + 10\% = 100\%\ \text{reliability}) \times 50\% + \\ &+\ 0.3\ (3 + 1 = 4\ \text{active resources}) \times 25\% + \\ &+\ 0.9\ (40\% - 20\% = 20\%\ \text{load}) \times 25\% \\ &= 0.80\end{aligned}$$

Discharge the Least Reliable

$$\begin{aligned}\text{Utility}_{\text{DLR}} &= 0.68\ (95.7\% + 1\% = 96.7\%\ \text{reliability}) \times 50\% + \\ &+\ 0.85\ (3 - 1 = 2\ \text{active resources}) \times 25\% + \\ &+\ 0.5\ (40\% + 20\% = 60\%\ \text{load}) \times 25\% \\ &= 0.6775\end{aligned}$$

6.3.4.2 Dynamic update of weights

Our approach predicts the behavior of each strategy by applying its changing operators to the architectural model. In other words, we apply to the stochastic model the same changes that a strategy would perform if it was selected. For example, if a strategy encompasses the discharge of a web-server as the changing operator, the system is modeled accordingly and the respective web-server is removed. Following this, we illustrate the modified models used to predict reliability and examine the utility calculations used to select the best strategy.

No adaptation

$$\begin{aligned}\text{Utility}_{\text{NA}} &= 0.6\ (95.7\%\ \text{reliability}) \times 50\% + \\ &+\ 0.55\ (3\ \text{active resources}) \times 25\% + \\ &+\ 0.7\ (40\%\ \text{load}) \times 25\% \\ &= 0.6125\end{aligned}$$

Enlist Server The modified architecture is illustrated in Figure 6.6, its generated Prism file is shown in Appendix A.1 and the utility calculations are shown afterwards.

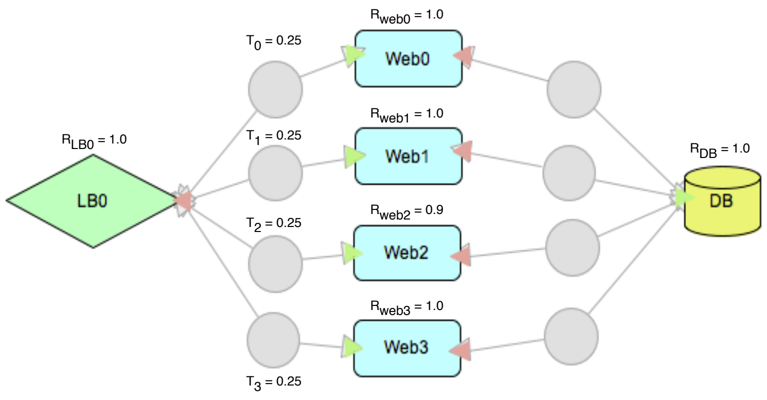


Figure 6.6: Architectural changes for the Enlist Server strategy

$$\begin{aligned}
\text{Utility}_{\text{ES}} &= 0.75\ (97.5\%\ \text{reliability}) \times 50\% + \\
&+\ 0.3\ (3 + 1 = 4\ \text{active resources}) \times 25\% + \\
&+\ 0.9\ (40\% - 20\% = 20\%\ \text{load}) \times 25\% \\
&= 0.675
\end{aligned}$$

Discharge the Least Reliable The modified architecture that is based on this strategy, is illustrated in Figure 6.7, its generated Prism file is in Appendix A.2 and the utility calculations are following shown.

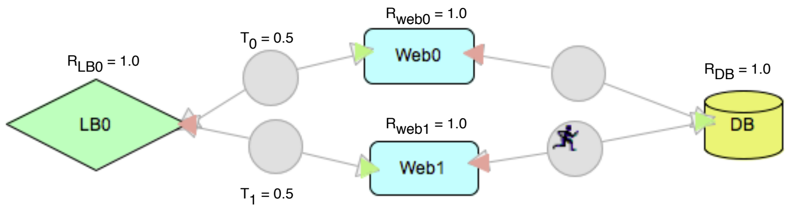


Figure 6.7: Architectural changes for the Discharge of the Least Reliable strategy

$$\begin{aligned}\text{Utility}_{\text{DLR}} &= 1.0\ (100\%\ \text{reliability}) \times 50\% + \\ &+\ 0.85\ (2\ \text{active resources}) \times 25\% + \\ &+\ 0.5\ (40\% + 20\% = 60\%\ \text{load}) \times 25\% \\ &= 0.8375\end{aligned}$$

The results of the utility calculated by both static and dynamic approaches are summarized in Table 6.4 and discussed in the following sub-section.

Table 6.4: Utility results from adaptation strategies

	No Adaptation	Enlist Server	Discharge Least Reliable
Constant Weights	0.6125	0.80	0.6775
Dynamic Update	0.6125	0.675%	0.8375

6.3.4.3 Discussion of Results

The obtained results from the above demonstration show that when a server is failing, the traditional self-adaptive approach based on static impact vectors, gives preference to enlisting a new server rather than discharging the failed one. This measure would add a new web-server, but not correct the failure.

On the other hand, our method performs reliability prediction to anticipate the impact of each strategy. This automated method allows architects to abstract from defining static vectors where it is hard achieve a precise impact and also to go beyond the static approach in unexpected or untested conditions, such as the above demonstration suggests.

One may argue that static impact vectors could be tuned or rearranged to account for this situation; however, another untested or unexpected condition may arise from this. These conditions are caused by the large state space of possibilities, which in this example one should account with three adaptation strategies, three runtime metrics and their utility values as well as preferences and impact vectors. Furthermore, our method relies on quantitative verification of reliability at runtime to predict the impact vectors and reduce the uncertainty at design time of estimating them. In conclusion, our approach makes better decisions by avoiding the constant weights required in the traditional self-adaptive approach.

6.3.5 Workload

We tested our approach with a realistic workload which can trigger different adaptations. More precisely, our workload is based on an Internet phenomenon, known as *Slashdot effect* or *flash crowd*. This phenomenon is characterized by a low-traffic website which may suddenly be inundated by visitors for a period of time due to, for example, a dramatic news announcement or alternatively, it may be redirected from a highly-visited website.

Our workload is depicted in Figure 6.8 and was patterned after the collection of realistic traffic from the event. The collected data of the event lasted twenty-four hours which we scaled down to one hour, keeping a similarity to the 'visit traffic pattern' as follows:

1. 1 minute of low activity;
2. 5 minutes of a sharp rise in incoming traffic;
3. 18 minutes of high peak requests;
4. 36 minutes of a linear decline in requests, also known as the ramp-down period.

6.4 Evaluation

For validation purposes, we compared three different approaches: no-adaptation, traditional decision-making and our dynamic update of adaptation weights. In more specific terms, the first approach entails a non-adaptive solution in which all the four servers are active and ready to respond to client requests (*i.e.*, 4 Servers). This approach aims to show the benefits of using adaptive solutions rather than a non-adaptive one. The second approach consists of human configured values

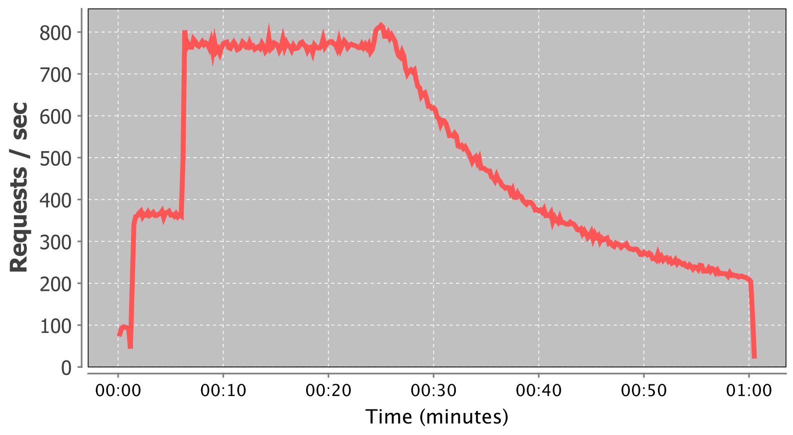


Figure 6.8: Request load of the *Slashdot effect*

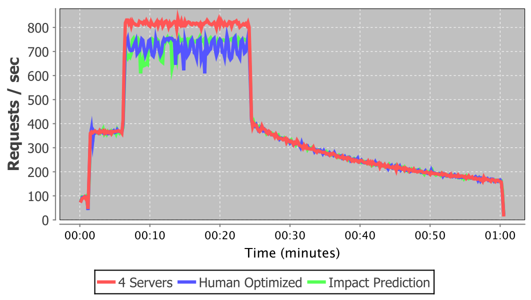
(*i.e.*, Human Optimized) to drive adaptations and shows the traditional decision-making algorithm. The third and last approach under comparison, includes the approach proposed in this paper and uses the runtime prediction of reliability to estimate the impact of each possible adaptation strategy (*i.e.*, Impact Prediction).

The experimental procedure consists of two testing scenarios: Control run and Fault-injection. In the former, we compare the approaches under normal conditions while, in the latter we inject faults to trigger adaptations.

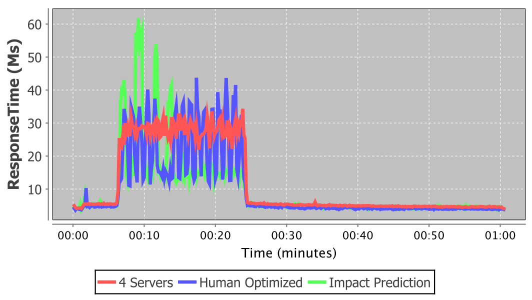
6.4.1 Control run

In this test, the system is in normal conditions without any injected fault or crash to ensure that both self-adaptive methods achieve their quality goals. The results are depicted in Figure 6.9 and as can be seen, there is a comparison between non-adaptive (4 servers), traditional self-adaptive (human optimized) and our self-adaptive proposal (impact prediction).

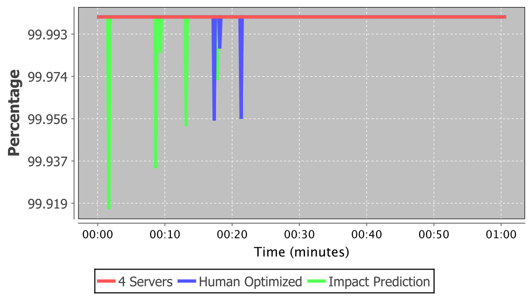
Figure 6.9(a) shows the throughput in number of processed requests (successful and unsuccessful requests) during each 10 second time-frame. As can be seen, the results show consistency between the tested scenarios and there is only a difference between them during the high-peak period of requests, when the scenario with more active servers will respond to more requests. Table 6.5 supports this claim by including a higher number of processed requests for the scenario with no adaptation and four active servers.



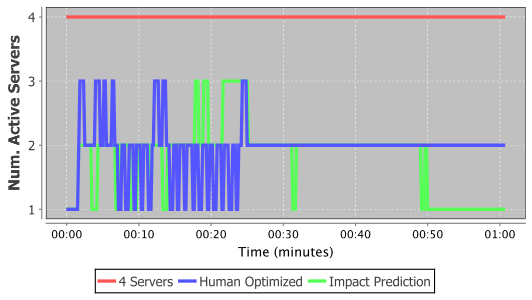
(a) Throughput



(b) Response Time



(c) Reliability



(d) Active Resources

Figure 6.9: Graphic results for the Control Run

Figure 6.9(b) shows the response time for each scenario in milliseconds with a granularity of ten seconds. An average response time is calculated for each ten second time-frame. An increase in the response time occurs during the high-peak period of requests, and returns to normal values in the ramp-down. Moreover, all the graph series have similar results, although between 6 and 24 minutes (the high peak period) both self-adaptive approaches have unstable results. This instability is due to the enlisting and discharging servers, although the results still remain within the threshold of acceptable response times (*i.e.*, below 2000 milliseconds).

Figure 6.9(c) shows the reliability through the rate of successful requests for each time-frame. The non-adaptive approach has a constant 100% of reliability throughout the test while the self-adaptive ones show some low peaks. These low peaks represent a very low number of failures (7 in the *human optimized*

Table 6.5: Control Run results

	N. Requests	**Availability** (%)	**VMs Hour**
4 Servers	1532989	100.0	4.0
Human optimized	1425479	99.999	1.9
Impact prediction	1437780	99.999	1.8

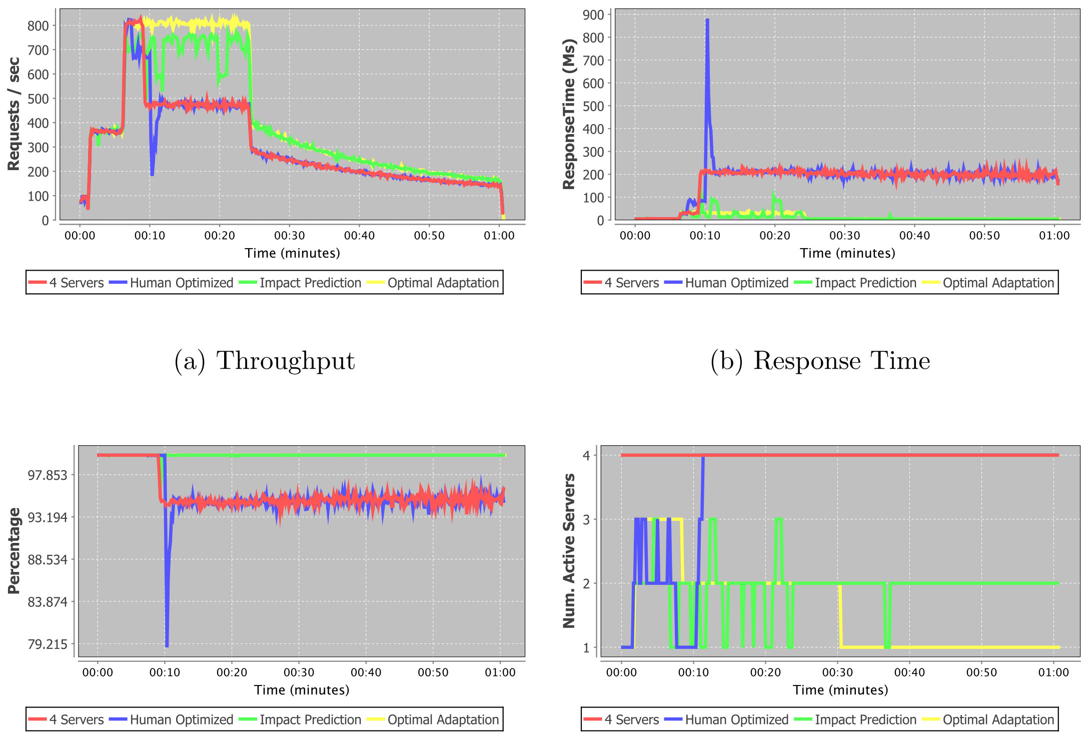
and 13 in the *impact prediction*) and are due to lost requests between switching servers on and off. Table 6.5 supports these statements by providing the long-run availability values and it can be seen that both self-adaptive approaches have five nines, while the non-adaptive does not register any failure.

The number of active servers during the experimentation is shown in Figure 6.9(d). The non-adaptive approach has a constant number of active servers, although self-adaptive ones have variations especially when there is a high demand for requests.

In conclusion, Figure 6.9 and Table 6.5 show that both self-adaptive approaches achieve similar results when compared with the most expensive non-adaptive solution. During this test, there are no significant advantages in applying our method for predicting the impact for each strategy and this was hardly the purpose of this experimentation. The set test is designed to show that both self-adaptive solutions achieve their adaptation goals, and result in similar and comparable results to the most expensive and available non-adaptive solution. However, the results for both the self-adaptive approaches are similar and thus it can concluded that runtime modeling and prediction of a quality attribute do not have an adverse effect on the performance of the system or the achievement of quality goals.

6.4.2 Fault injection

Human operators are often considered the weak link and the proportion of errors that can be attributed to people ranges from 0.1% to 30%, depending on whether the operator is handling simple routine operations or undergoing a high level of stress [Kirwan, 1994]. In view of this, we set up an experiment that injects a fault in a PHP file that corresponds to a mistake introduced by the developer. The fault consists of a delay introduced in each request that leads to service degradation by increasing the time each request is resolved by between 1.5 and 2.5 seconds, following an uniform distribution. The rationale behind the specified values is that a request is regarded as unsuccessful if it takes more than two seconds to be resolved. Thus, the introduced delay allow some requests to be resolved within the time regarded as successful and others to exceed the requisite amount of time leading to failures.



(a) Throughput

(b) Response Time

(c) Reliability

(d) Active Resources

Figure 6.10: Graph results for the Fault Injection experiment

The injected fault only affects one server and is introduced 10 minutes after the start of the experiment. Figure 6.10 shows the results for this experiment in which we compared the best of non-adaptive (4 servers); these include both self-adaptive approaches and a run that we consider to be the *optimal adaptation*.

This *optimal adaptation* assumes that the system knows when and where the failure will occur, so it proceeds by disabling the failing server and enlisting a spare one. This 'ideal' adaptation is considered to be unrealistic in the real world, since it assumes knowing *a priori* when and where to apply a recovery action. The goal of this test is to keep a record of the best possible adaptations and identify how close other adaptation methods get to this ideal adaptation. Table 6.6 includes a complete list of the performed tests.

Table 6.6: Fault injection results

	N. Requests	**Availability (%)**	**VMs Hour**
4 Servers	1110584	95.752	4.0
Human optimized	1096751	95.682	3.6
Impact prediction	1417786	99.981	1.9
Optimal adaptation	1523411	99.999	1.6

The throughput results are given in Figure 6.10(a) and show an abrupt fall in the number of processed requests at 10 minutes due to the introduction of a delay. It can be seen that the non-adaptive (4 Servers) approach cannot recover from the 'failing behavior', and leads to an increase of response time and a decline in reliability as shown in Figures 6.10(b) and 6.10(c), respectively.

In both *4 servers* and in *human optimized* approaches, the number of processed requests shown in Figure 6.10(a) falls sharply. The reason for this is that the load-balancing policy distributes the same amount of work among the various active servers. To keep equality among the servers, the load-balancer waits for all the responses before distributing a new set of requests. For this reason, if the failing server remains in the pool, the load-balancer has to wait for its response which causes a delay and, thus, reduces the number of processed requests.

With regard to the *human optimized* run, Figures 6.10(a) and 6.10(c) show an abrupt fall in the number of processed requests as well as in reliability. Figure 6.10(d) illustrates its adaptation process which consists of increasing the number of resources to cope with the introduced delay. However, since it just enables more servers and keeps the failing server active, the reliability and performance will always be affected, since the failing server has to respond to requests and the others will have to wait for it.

On the other hand, in the *impact prediction* there is clearly also a fall in reliability and throughput. However, by predicting the impact of each adaptation strategy, it first decides to discharge the failing server and then adds more resources to cope with the demand for requests, as seen in Figure 6.10(d) and demonstrated by the utility calculations set out in Section 6.3.4.2. As a result, our method quickly triggers proper adaptations to cope with this kind of erratic behavior, by providing a high number of processed requests and a low number of failures throughout the rest of the experiment.

These results confirm that unexpected or untested conditions may have a negative effect on the achievement of quality goals when using constant weights are used to trigger adaptations. Moreover, Table 6.6 suggests that the *impact prediction* has good overall results with lower cost and better availability than the other approaches. Moreover, it can also be confirmed that runtime modeling and prediction of quality attributes positively influence decision-making in unexpected or untested conditions.

The consequences of injecting a fault are only undetectable in the *optimal adaptation*. This is because the presence of the fault is known beforehand, and thus appropriate measures are taken to repair the fault before it can lead to a failure. This means we can achieve an optimal result for the self-adaptive system under fault injection. However, this scenario might be unrealistic, since we assume that the system knows when and where the fault will be injected. In Table 6.6 it can be observed that our approach obtains excellent values regarding availability and costs which are close to the ideal and unrealistic results of the *optimal adaptation*.

In short, both *human optimized* and *impact prediction* solutions can be recovered from the erratic behavior, although the chosen adaptation strategies differ. More precisely, both adaptive approaches have configured the enlisting server strategy to improve availability (as outlined in Section 6.3.3.4). Although this assumption may have a positive effect in most cases, it is not always true as shown by this experiment. Hence, when the failure occurs, the *human optimized* solution selects the enlisting server strategy to increase availability, as shown in Figure 6.10(d), until it reaches the maximum size of the server pool. If our setup scenario had more available servers, the *human optimized* approach would have reached the maximum pool size too. However, our approach predicts the impact of choosing each strategy and when the failure occurs, it decides to discharge the failing server. As a result, it can be concluded from this experimental procedure that by predicting the impact of each strategy, the self-adaptive system is able to make informed decisions and achieve the desired adaptation goals.

6.4.3 Effectiveness and scalability of impact prediction

Formal approaches that require an analysis of large state spaces may often be time-consuming, and may lead to considerable overheads. To address this issue, we optimized our approach to complete the execution within the ten-second time window. This requirement ensures that there is never a different overlapping analysis. Table 6.7 shows the time that each prediction takes to complete. It can

Table 6.7: Time, in seconds, taken to predict the impact of each strategy

Number of Servers	Mean	Std. Dev.	95^{th} Percentile
4 Servers	0.15	0.14	0.27
25 Servers	0.22	0.25	0.44
50 Servers	0.40	0.26	0.50
75 Servers	0.55	0.39	0.79
100 Servers	0.79	0.59	1.15

be observed that in the case-study with four active servers, each analysis takes, on average, $0.15\,s$ and the 95th percentile takes 0.27 s. Our approach generates and solves a stochastic model for each strategy, so the total time in the analysis is obtained through $Num.\ Strategies \times Time\ spent\ on\ prediction$. In our case-study, there are three strategies, which means analysis usually executes in less than one second ($0.27s \times 3 = 0.81s$), well below the ten-second requirement.

In an attempt to evaluate scalability, we tested our implementation for an increasing number of servers, and the results are illustrated in Figure 6.11. We

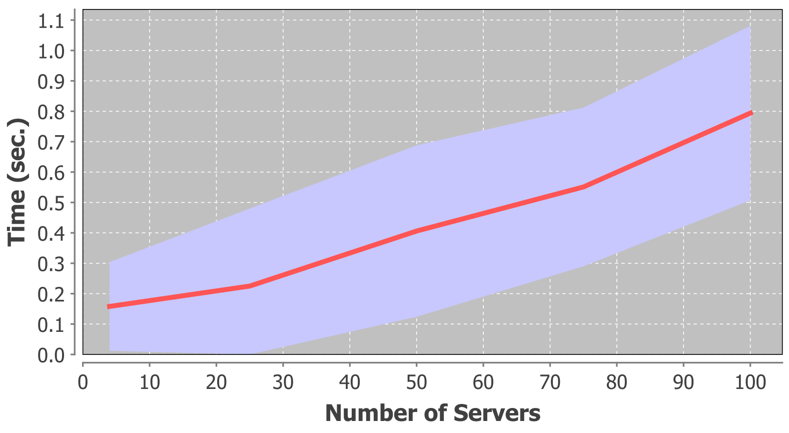


Figure 6.11: Scalability of our approach regarding the number of Servers

ran each experiment 30 times and collected the mean and the standard deviation which are shown in the diagram. We can observe an increase in the time spent in each prediction, but the analysis scales well for a system with up to 100 active servers. If there is a scenario with 100 servers, our approach would take less than 10 seconds to execute, given the prediction of the three adaptation strategies ($1.15s \times 3 = 3.45s$). This would be the case not only for the 95^{th} percentile but also for the maximum time observed.

6.4.4 Discussion

In the control run experiment, we tested the two self-adaptive solutions against a non-adaptive one. The goal was to determine whether both self-adaptive systems achieve the non-functional requirements, and maintain high availability while reducing the usage of computational resources. The results show that both self-adaptive solutions achieve similar results with the most expensive and highly available non-adaptive solution. However, self-adaptive approaches outperform the non-adaptive one by using 50% less computational resources.

When failures occur, the experimental results show that the two self-adaptive approaches adopt different strategies. In particular, the *human optimized* approach uses static weights while the *impact prediction* approach uses stochastic models to predict the failure behavior of each adaptation strategy. As a result, our method selects the best strategy to recover from failures, and achieves an increased performance and availability, while reducing the usage of computational resources.

6.5 Related Work

Self-adaptive systems are able to adjust their behavior in response to their perception of the environment and the system itself [Lemos *et al.*, 2013]. These systems are usually implemented through the MAPE-K approach defined by the International Business Machines (IBM) Corporation in 2004 [IBM Corp., 2004]. The MAPE-K is a short name for Monitor, Analysis, Plan and Execute tasks, all with a shared Knowledge-base. To be more specific, a self-adaptive system *monitors* the environment and the system itself to *analyze* whether an adaptation is required or not to achieve the desired goals. In case of being necessary, a course of action is *planned* and *executed* in the target system to change the current behavior and achieve the desired quality goals. Communication between different adaptation phases is conducted through a shared *knowledge*-base that abstracts the system, containing data, models, decisions and behavior, enabling separation of adaptation responsibilities and allowing their coordination.

Self-adaptive systems have been an interesting focus of research study due to their ability to adapt and modify the behavior leading to a multitude of practical applications, like self-driving cars, self-maintainable software and self-ruling systems. Salehie *et al.* [Salehie & Tahvildari, 2009] present a survey article about the landscape of research, taxonomies, gaps, and future challenges in self-adaptive systems. They consider the adaptation process as a concept that deserves attention in future research challenges. One of the most important challenges they highlighted is the assurance that an adaptation is going to have a predictable impact on the functional and non-functional aspects of the system. To this end, we following describe the approaches that perform quantitative prediction of non-functional attributes or verification of correctness in self-adaptive systems.

Quantitative verification is a technique to calculate the likelihood of the occurrence of certain events during the execution of the system. The benefits of having this verification at runtime to support software adaptation are discussed by Calinescu *et al.* [Calinescu *et al.*, 2012]. They state that by employing modeling techniques at runtime (*e.g.*, predict requirements violation, plan recovery from such violations and verify correctness in the adaptation steps employed in recovery) we may obtain more dependable self-adaptive systems.

The work of Gallotti *et al.* [Gallotti *et al.*, 2008] proposes an approach to generate stochastic models to assess reliability and performance from Activity Diagrams described in Unified Modeling Language (UML). In short, their approach takes as input a formal representation of service composition drawn as a UML Activity Diagram along with a specification of quality properties, such as response time or failure rate. The approach interprets the draw and creates an intermediate representation before generating a stochastic model to be solved by Prism [Kwiatkowska *et al.*, 2009], a model checking tool. The interpretation of the draw cannot be standardized for every UML Activity Diagram tool, since their representations differ in small details. This work differs from ours, since we focus on Architectural Description Languages (ADLs) than UML and we also propose a formal notation to standardize the translation from an architectural model complying with the ISO/IEC/IEEE 42010 Standard [ISO/IEC/IEEE, 2011] to a stochastic model. In addition, we perform reliability prediction at runtime and show its effectiveness by conducting a performance assessment, while Gallotti *et al.* address these as future work.

Cámara *et al.* [Camara & de Lemos, 2012] propose an approach that models the behavior of a self-adaptive system with regard to trustworthy service delivery. In more detail, the authors model the adaptation behavior of the system to obtain levels of confidence regarding the resilience of each adaptation. The effectiveness of their approach is outlined through an experimentation similar to ours, using Rainbow and Znn.com as self-adaptive solution, respectively. The results show that both outcomes from the proposed modeling approach and from the running system are close validating their work. The work of Cámara *et al.* [Camara & de Lemos, 2012] differs from ours in the aspect that we automatically generate and solve stochastic models at runtime to support the adaptation manager by deciding which is the best strategy to attain the desired quality goals.

Zheng *et al.* [Zheng *et al.*, 2008] applied Kalman Filters to model and track performance that can be used to evaluate end-to-end response times, utilization of resources and also to estimate performance parameters. Their work has been applied to autonomic computing to empower decision-making capabilities. The approach was tested in a scenario of a cluster of servers and the results show that it efficiently maintain service level and avoid system overload. In more detail, it takes into consideration disturbance changes such as the number of users, software aging or requests with modified resource demands.

Filieri *et al.* [Filieri *et al.*, 2011] explore models and system adaptations to meet a particular target reliability through a control theoretical approach. In more detail, they keep alive a model of the application at runtime which expresses reliability concerns through a DTMC. This model is continuously updated at runtime and, in a control-theory viewpoint, is viewed as the input variables to the controlled system. They consider self-adaptation at the model level, where possible variant behaviors are evaluated and the selected changes are then transferred into the running implementation. Their approach bypasses the decision-making process of the self-adaptive system and rely upon only one adaptation goal: Reliability.

Any of the aforementioned studies that address quantitative prediction or verification at runtime are likely applicable to a self-adaptive system to enrich its decision-making process. This enrichment is made by replacing constant adaptation operators by predictions on the quality outcome or by assuring correctness for each adaptation. We argue that the resulting system will be able to make informed decisions about the impact on the quality dimensions in unexpected and unanticipated situations. These studies address uncertainty by keeping a model alive which is constantly updated with runtime properties. However, the construction of the model is the responsibility of the designer or the engineer, increasing development effort, time to deliver and cost. This is where our work augments the current research field by automatically generating probabilistic models at runtime reducing effort and modeling time, at the same time that accounts with structural changes in the architecture such as changing architectural styles or the addition of new components.

The work that closely relates to ours is QoSMOS (Quality of Service Management and Optimization of Service-based systems) proposed by Calinescu *et al.* [Calinescu *et al.*, 2011]. QoSMOS assures that QoS is delivered by adaptive systems in an equally adaptive and predictable way. QoSMOS support self-adaptation of service-based systems by choosing an optimal strategy through prediction of QoS at runtime. In short, they combine several existing techniques, such as the formal specification of QoS by using temporal logic, generation of stochastic models to evaluate reliability and performance, Bayesian-based parameter adaptation by exploiting KAMI [Epifani *et al.*, 2009] and a tool to support the planning and execution phases of the system adaptation. Our work closely relates to QoSMOS since both works assure quality of the system adaptation, perform quality prediction at runtime and evaluate the approach scalability and performance. Regarding the difference of both works, QoSMOS takes as input BPEL (Business Process Execution Language) models of service orchestration focusing on only service based systems, while our work uses Architectural Description Languages (ADLs). ADLs allow to specify a wider range of systems, reveal the topology or structure of the whole system and define architectural styles. In addition, we propose a formal notation to translate from the ADL to the stochastic model providing a generic solution that can be applied to other quality attributes or ADLs. Regarding QoSMOS scalability and evaluation efficiency, it cannot be applied to large scenarios due to the exhaustive quantitative model checking in the Analysis phase. In more detail, QoSMOS evaluates six different PCTL rules (4 for reliability and 2 for performance) while our approach assesses one rule that expresses the reliability of the system. This limits QoSMOS application to only systems in which time efficiency is not a problem, while our approach can be applied to those which have strict time requirements. The evaluation of QoSMOS is obtained through a theoretical case-study of a TeleAssistance scenario while our approach has been implemented and running on an actual case-study of a news infrastructure system. This case-study allows to validate our approach with an application example, as well as to compare results from traditional approaches with ours.

6.6 Summary

Self-adaptive systems are becoming more common in our daily tasks, although this is sometimes unnoticed, until a failure occurs. When these systems affect human lives, like self-driving cars, they only need to make one wrong decision to fail and become discredited by their users and cease to be dependable, especially when human lives are at risk. Bearing this in mind, we applied our automated method to predict reliability from a software architecture to enhance the planning phase of self-adaptive systems. Our goal was to show the applicability of our approach and its effectiveness in improving real world issues.

Evidence of the limitations of current decision-making approaches and the validity of our method was obtained by conducting a realistic experiment based on a news infrastructure hosted in a cloud environment. The experiment which is discussed in Section 6.4.2, confirms that traditional decision-making approaches (*i.e.*, human optimized) fail to select the best strategy in unexpected or untested conditions and this leads to the degradation of the delivered service. In addition, the same experiment enabled us to conclude that our approach (*i.e.*, impact prediction) can recover from erroneous behavior, by validating the use of quantitative prediction methods at runtime. This method improves the ability to reach quality goals in unforeseen circumstances, while maintaining a similar ability in known ones.

Our approach entails the generation of stochastic models and the means of solving them, which are tasks usually seen as time-consuming and inefficient. It can be concluded from the experiment in Section 6.4.2 that our approach (*i.e.*, impact prediction) performs as well as other approaches by providing similar values of resolved requests, throughput and resource consumption. Furthermore, we conducted an experiment to assess the performance and scalability of our approach. The obtained data shows that our method performs under one second and if the system could scale to one hundred web-servers, our approach would still fulfill the performance requirements.

To conclude, the experimental work covered in this chapter can be of value to the self-adaptive community by employing a method that allows a correct and context-sensitive strategy to be adopted to achieve the specified quality goals. Given that self-adaptive systems are recognized as a solution for dealing with highly complex environments, we expect our approach to further improve the current solutions in unforeseen circumstances.

Chapter 7

Modeling the Failure Pathology of a Software Component

The methods and techniques to predict, estimate and analyze reliability used by classical reliability studies [Cheung, 1980; Goševa-Popstojanova & Trivedi, 2001; Reussner & Heinz W., 2003; Wang *et al.*, 2006] have immediately attracted the attention of the software research community owning to their potential applicability to software architecture. If system properties can be assessed in the early stages of the development cycle and weak architectural points identified, these can be corrected and thus reduce the number of late-detected problems.

The potential benefits both in delivery time and budgeting have been strong research drivers in this area. However, classical reliability prediction methods pose their own intrinsic difficulties [Gokhale, 2007], namely the assignment of realistic reliability values for components before they have actually been built. Moreover, these methods assume that when an error arises it manifests itself as an application failure or that the error propagates to the application output [Filieri *et al.*, 2010]. This assumption means that if an error occurs in one of the components, the whole application fails without taking into account the likely possibility of masking, recovery, or tolerance of that error.

There are also methodological issues arising from the newness of this research area as realistic case-studies with actual failure data have not yet been disseminated. The software architecture reliability research community would greatly benefit if convincing architectures and associated failure data were made public, since this would allow the comparison of results and validation/benchmarking of different approaches.

To address the above problems we propose an approach that models reliability by taking account of the failure mode discussed by Avizienis *et al.* [Avizienis *et al.*, 2004]. Our model is extracted from an actual software architecture and includes error masking, error propagation, failure recoveries and multiple-failure modes. The goal is to express the failure pathology of a software component in order to obtain more realistic and accurate stochastic models.

To overcome the lack of actual case-studies and failure data, we performed an experiment that not only serves to validate our approach, but also makes a contribution to the research community. The experimental results were obtained by injecting faults into a running system comprising a freely available, widely used virtualization system for cloud-based services(Xen). This experimental testbed and its actual failure data can thus be used by other research studies to test, validate and benchmark different reliability methods.

In summary, this chapter encompasses the following contributions:

1. A stochastic model expressing the failure pathology of a software component accounting for errors, failure recoveries, propagation, error masking and different failure modes.

2. An actual case-study encompassing realistic failure data.

To achieve the proposed approach and implement the case-study for testing and validation purposes, this chapter is structured as follows. Section 7.1 outlines our approach to model the failure pathology of a software component. Our case-study is discussed in Section 7.2 and tried out in Section 7.3 where we test the validity of our approach. The related work is outlined in Section 7.4, before Section 7.5 concludes this chapter by summarizing the proposed approach, its contributions and validation.

7.1 Modeling

Reliability prediction from the perspective of a software architecture description, has been extensively studied and several methods have been employed [Gokhale, 2006; Goševa-Popstojanova & Trivedi, 2001]. Our approach extends the reliability methods by including the failure pathology of a software component. More precisely, our approach encompasses error masking, error propagation, failure recoveries and multiple-failure modes. We propose to model the system behavior through a Discrete-Time Markov Chain (DTMC) as described in Section 2.4.1 and illustrated in Figure 7.1.

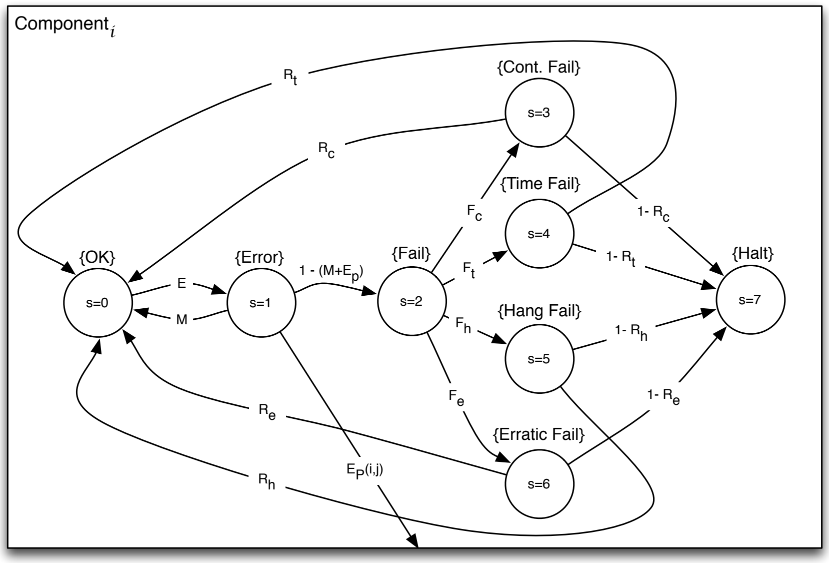


Figure 7.1: DTMC illustrating the failure behavior of a single software component

A software component represents a unit of computation and can be in either one of the seven states depicted in the modeled DTMC:

- OK – the component is not in the presence of an error and executes as expected. When an error occurs with probability E the control of execution exits the OK state ($s = 0$) and enters in the error state ($s = 1$).
- Error – represents when a fault is activated and this leads the component to enter in an error state. When an error occurs, one of the following outcomes can occur:

  - the error is masked with probability M and returns to the OK state ($s = 0$). An error can be masked through fault tolerance mechanisms, including error detection or recovery techniques which bring the system to a correct state;
  - it can enter a failed state ($s = 2$);
  - the error can be propagated to component j with rate $Ep_{(i,j)}$.

- Fail – this state ($s = 2$) denotes when an error is activated, leading to a failure. When a component fails, it can be categorized as content, timing, hang or erratic [Avizienis *et al.*, 2004] with rate probability F_c, F_t, F_h, F_e, respectively.

- Content, Timing, Hang and Erratic Failure states – describe the multiple failure modes that follow the categories of the failures proposed by Avizienis *et al.* [Avizienis *et al.*, 2004]. After entering in one of these failure modes, the system can recover due to, for example, the implementation of failure handling techniques. The recovering rates from the failure states content ($s = 3$), timing ($s = 4$), hang ($s = 5$) and erratic ($s = 6$) to the state OK ($s = 0$) are given by R_c, R_t, R_h, R_e, respectively.

- Halt – this state ($s = 7$) illustrates the complete stop of the system and emulates a failure that leads the system to a crash without the possibility of being recovered without human intervention.

To model the failure mode of a system, we use an absorbing DTMC which encompasses two final states with self-loop transitions defining the successful (S_c) and failure states (s_f). The rationale behind this approach is that each task is processed through the various states in the model and reaches one of two conditions: a successful (s_c) or a failure state (s_f).

To obtain a probabilistic quantification of the modeled system reliability we specify a property in the Probabilistic Computation Tree Logic (PCTL) [Kwiatkowska *et al.*, 2007]. This property computes the probabilities from the initial state $\overline{s}$ until the successful absorbing state ($state = s_c$) has been reached.

To model the interactions of different software components, several studies adopt a user-oriented approach [Cheung, 1980] [Goševa-Popstojanova & Trivedi, 2001] which assumes a sequential order in the execution. However, a system may encompass different architectural patterns in which components may execute in parallel or concurrently. To model these kinds of behavior, Wang *et al.* [Wang *et al.*, 2006] proposes a method to solve a DTMC according to each applied pattern.

This model is novel approach since it represents the failure pathology of a component. It extends the work of Cheung [Cheung, 1980] and has similarities with the study of Filieri *et al.* [Filieri *et al.*, 2011] by including error propagation and multiple failure modes, but differing in the comprehensiveness of the model. Our proposal is a reliability model that supports the comprehensive failure mode specified by [Avizienis *et al.*, 2004] comprising error, masking and propagation to other components, multiple failure modes and possible recoveries.

In the following section we provide a detailed examination of the case-study used to validate the method described above.

7.2 Case-study

To validate the accuracy of the model proposed in the previous section, we implemented a realistic case-study to compare the predicted reliability results with those obtained from real-world experiments. The case-study is based on an actual cloud infrastructure (not a simulation model) and deploys a set of HTTP servers responding to client requests. To ensure that our experiments were reproducible, we adopted a widely known, freely available virtualization solution, the Xen hypervisor [Barham *et al.*, 2003], which is used in services like the Amazon Elastic Compute Cloud (EC2), RackSpace Mosso or the CloudEx [Qian *et al.*, 2009].

Figure 7.2 illustrates our tested scenario where Xen is depicted above the hardware layer and under the instanced virtual machines (VMs). The Dom0 is a privileged VM responsible for managing the virtualization environment and has direct access to the hardware.

In this case-study, we used two guest virtual machines (VM1 and VM2) re-

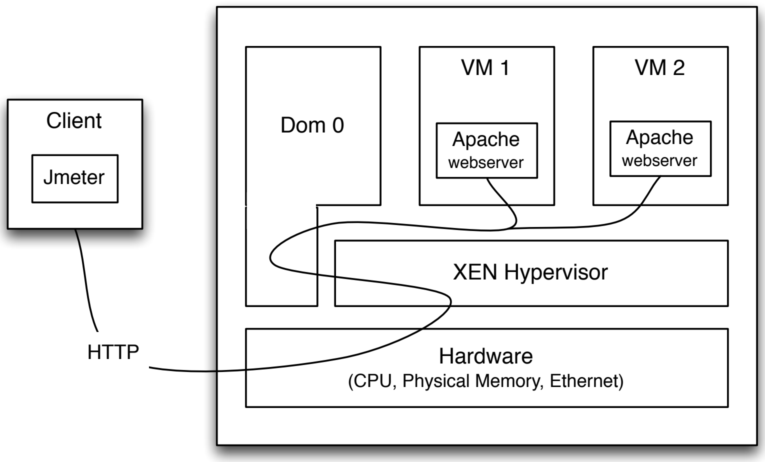


Figure 7.2: Diagram of the case-study

sponsible for responding to client HTTP requests. Both VMs have the same system image and the same workload. Our goal is to inject faults into one of the VMs and check whether the errors are propagated to the other one, i.e. if the isolation between both VMs has been compromised.

The edge depicted in Figure 7.2 from the Client to the VMs that passes through the different components, illustrates the connection used for communication between the client and servers. In addition, this edge also illustrates the system dependencies among the architectural elements. The client sends a request to a physically separated machine, where the case-study resides. This machine receives the request through its ethernet card (hardware), and is interpreted by the Xen hypervisor which is responsible for redirecting the requests to the appropriate VM. Through the whole process, Dom0 is able to access the hardware and possibly monitor or modify this interconnection.

In this case-study, the client is an external independent computation resource which performs HTTP requests concurrently to each VM through a Jmeter application. Each request is processed by the Apache webserver by calculating a SHA1 hash using as input a generated file with a fixed size and content, which always results in the same hash unless an error occurs. This hash is then reported in the HTML response which can be assessed by the client whether it is the expected one or to conclude that a failure has occurred. In this experiment we defined the size of the generated file to be 1024 MB. The reason for this figure is that small file sizes would result in a short life span for processing the response which might limit the effectiveness of an introduced error. In light of this, we opted for a large file size to increase the processing time to exercise CPU and memory, thus allowing the introduced error to be propagated and become noticeable.

This experiment includes a workload to simulate several HTTP clients performing requests to the available webservers. Thus, we configured Jmeter to use 10 clients that start by making requests during a 30 second ramp-up period and then perform a request at each 600 milliseconds with a 100 millisecond deviation pause between them. As a result, each experiment run lasts approximately 420 seconds (7 min) of which 330 seconds (5.5 min) comprise the time when the workload is executed, and the remaining 110 seconds are divided between the setup period before the experiment (*i.e.*, to launch probes and prepare the fault injection tool) and the clean-up period at the end (*i.e.*, extract logs, clean temporary files and restart machines).

7.2.1 Fault Injection

Estimating error propagation and failure occurrence probabilities can be quite a difficult task due to the number of factors involved and to the inherent randomness of operational profiles, inputs, failure types, etc. Thus we performed fault injection to artificially generate errors in the system. The goal was to collect relevant failure data and identify elements that are more prone to failure due to error propagation.

A fault is injected during the execution of the workload in an interval between 30 seconds and 4 minutes, which is randomly chosen following a uniform distribution. This interval guarantees that the fault is injected after the ramp-up period and that the target system is working at its nominal throughput. It also provides enough time before the end of the experiments for the fault to be propagated.

The faults injected emulate transient hardware faults, by flipping bits in one of the available registers. Both the bit to flip and the register to target are chosen randomly following a uniform distribution.

For this case-study, we injected faults in the Dom0 and in one of the VMs.

The goal of the former injection is to assess the faulty behavior of the Xen and the VMs when a fault is introduced in the hypervisor. The latter enables us to determine if the isolation between VMs will be compromised if an error occurs in one of the Virtual Machines.

In summary, the fault injection experiment is performed at the client side by first sending a HTTP request to one of the Virtual Machines. The webserver in the target VM is responsible for processing the request by creating a file with a fixed size and content. After creating the file, the server calculates its SHA1 hash and sends the computed hash as the response to the client. While the server is processing the request, we inject a fault by flipping a bit in one of the CPU registers. If the error is activated, the content of the file or the response may change, leading to a failure. For this reason, the client is expecting a particular hash, since it knows the size and content of the file; thus the hash should be the same unless an error has occurred. As a result, the client can determine if the injected fault has led to a failure or the internal faulty behavior has been masked.

7.2.2 Failure Classification

When modeling multiple failure modes, we require a classification of the different type of failures that an error can lead to. With this in mind, we adapted the failure classification from Avizienis *et al.* [Avizienis *et al.*, 2004] to our case-study as follows:

- Content – a failure occurs when the content of the received information deviates from what is expected and correct. Since in our case-study, we generate a file for each request with always the same size and content, its hash will invariably be the same. Thus, whenever we receive a response with a different hash, it can be concluded that a content failure has occurred.

- Timing – occurs when the amount of time between sending a request and waiting for a response falls outside of a reasonable interval, leading to an early or late service failure. In this case-study, we assume that if a request is not answered between 2 and 15.78 seconds, it is considered to be a timing-failure. These values have been extracted from the actual testbed after a large number of runs without fault injections. The maximum and minimum recorded response times were defined to be the upper and lower bounds of the acceptable time interval.

- Halt – this failure mode arises when there is an unexpected absence of system activity. In our experiment, we account for a halt-type failure when a connection is dropped, suddenly closed or the server stops responding.

- Erratic – when the service is not halted but suffers a disruption in the delivered service, such as an arbitrary correct or incorrect response. In our case-study, we assume that a failure is erratic if both the response content and timing deviates from what is expected or the connection is still alive, but experiencing anomalies.

To validate the reliability of the assessment method proposed in Section 7.1, we conducted a set of experiments which are outlined in the following section.

7.3 Experimentation

To prove the validity of the model proposed in Section 7.1, we conducted a realistic experiment to compare both the predicted reliability and what was actually obtained. In this section we set out the gathered results, examine the validity of our approach and discuss the results of the experimentation.

7.3.1 Gathering Failure Data

To validate our approach, we collected failure data by injecting faults into the system. In particular, two means were employed: directly into the VM1 and in the Dom0. The goal of the former is to obtain relevant failure information about the impact of a transient hardware fault into the virtualized system. The injected fault only targeted one of the VMs (*i.e.*, VM1), leaving the other intact (*i.e.*, VM2). The purpose of this is to detect whether there has been any breach in the isolation between both VMs which is usually assumed as guaranteed. The aim of the latter is to study the effect of introducing faults directly into the hypervisor.

7.3.1.1 Injection in VM1

The Xen hypervisor supports two types of virtualization: Paravirtualization (PV) where guest OSes run efficiently without requiring virtual emulated hardware and Hardware Virtual Machine (HVM) (also known as Full virtualization) in which guest OSes require the complete hardware emulation to run, such as BIOS, USB controllers or graphical adapters. The fault injection experiment in VM1 takes account of these types of virtualization by specifically making separate runs.

Table 7.1 shows the results of the experiment with the HVM virtualization type where we performed 1028 runs and injected a fault in each one. In these runs we observed 876 injected errors that had been masked and 152 that had led to failure. Of those 152 failures, 4 manifested as content, 1 as timing, 147 as hang and 0 as erratic failures. Moreover, recovery was possible in every content and timing failure. However, in 22 runs that were hang failures that could not be recovered, which left the system in a completely inoperative state.

Furthermore, the results show that of the 1028 faults injected in VM1, none affected VM2. Thus, the experiment shows that the integrity of the isolation layer between virtual machines is preserved by flipping bits in CPU registers for the HVM virtualization type.

With regard to the results of Paravirtualization (PV), Table 7.2 depicts the failure rate obtained after 968 fault-injections. In those 968 runs, we observed 876 errors that had been masked and 92 injections that had led to failure. More specifically, 7 failures were assigned as content type, 0 as timing, 85 as hang and 0 as erratic. The results show that all the content failures had been successfully recovered, while 6 hang failures led the system to become unresponsive.

The results also show that the injected faults in this virtualization type did not affect the isolation layer between the Virtual Machines (VMs). This allows us to conclude that Xen is capable of confining error propagation between different guest VMs even when one VM is subject to fault-injection, in both virtualization

Table 7.1: Hardware Virtual Machine (HVM) results

	HVM			
	VM 1		VM2	
	Failure	Recovery	Failure	Recovery
Content	4	4	0	0
Timing	1	1	0	0
Hang	147	125	0	0
Erratic	0	0	0	0
Total	152	130	0	0
Total Non-Failures	876		1028	
Total Halts	22		0	
Total Injections	1028			

types (PV and HVM)

7.3.1.2 Injection in Dom0

The privileged virtual machine Dom0 is responsible for managing the virtualization environment and has direct access to the hardware. Thus, we injected faults in this privileged VM with the aim of analyzing the system behavior and identifying error propagation in the hosted VMs. To this end, we injected faults into the following kernel extensions:

- Qemu-system-i386: used by Xen to enable the dom0 to access a virtual disk;
- Xenwatchdogd: allows actions to be triggered when certain guest VMs are detected as having crashed.

Table 7.2: Results of Paravirtualization (PV)

	PV			
	VM 1		VM2	
	Failure	Recovery	Failure	Recovery
Content	7	7	0	0
Timing	0	0	0	0
Hang	85	79	0	0
Erratic	0	0	0	0
Total	92	86	0	0
Total Non-Failures	876		968	
Total Halts	6		0	
Total Injections	968			

Table 7.3 shows the results of injecting faults into the Qemu kernel extension. We flipped bits in 101 runs in which both guest VMs had been affected. In those 101 runs, VM1 and VM2 masked 94 errors while 7 led to failures that brought the system to a halt. Only one type of failure was registered (hang) and the guest VMs were affected in the same way.

Table 7.4 displays the failure rate data in the XenWatchdog extension. It should be noted that after the 73 injection runs, 56 were masked errors in both VMs. After the 17 triggered failures, none could be recovered. As in the previous experiment with the Qemu extension, only one type of failure was manifested, the hang-type failure, and again the injected faults affected both guest VMs in an equal way.

In short, the results show that errors can be propagated from Dom0 to guest Virtual Machines, leading to a failure of a single type – the hang failure. It was also observed that when a failure occurs it cannot recover and this puts the system

Table 7.3: Fault Injection in the Dom0 in the Qemu extension

	Qemu			
	VM 1		VM 2	
	Failure	Recovery	Failure	Recovery
Content	0	0	0	0
Timing	0	0	0	0
Hang	7	0	7	0
Erratic	0	0	0	0
Total	7	0	7	0
Total Non-Failures	94		94	
Total Halts	7		7	
Total Injections	101			

in a completely inoperative state. The observation that both guest VMs always display equal behavior can be explained by the fact that they share the same resources, system image, workload and are thus subject to the same fault-load.

7.3.2 Validation

Two methods are put forward to validate the reliability model: validating the model for each software component and then, confirming its accuracy regarding error propagation.

7.3.2.1 Single component validation

In this experiment, we applied the values collected from the fault injection to the VM1 while taking account of both virtualization types. Our goal is to validate

Table 7.4: Fault Injection in the Dom0 in the XenWatchdogd extension

	XenWatchdogd			
	VM 1		VM 2	
	Failure	Recovery	Failure	Recovery
Content	0	0	0	0
Timing	0	0	0	0
Hang	17	0	17	0
Erratic	0	0	0	0
Total	17	0	17	0
Total Non-Failures	56		56	
Total Halts	17		17	
Total Injections	73			

the proposed model and determine which virtualization type is more reliable. Table 7.5 depicts the failure data collected from the experimental results shown in Tables 7.1 and 7.2.

Table 7.5: Modeling Parameters

		M	F_c	F_t	F_h	F_e	R_c	R_t	R_h	R_e
HVM	VM1	0.8522	0.0263	0.0065	0.9671	0.0	1.0	1.0	0.8503	0.0
PV	VM1	0.9050	0.0761	0.0	0.9239	0.0	1.0	0.0	0.9294	0.0

M - Masking F_c - Content Failure F_t - Timing Failure

F_h - Hang Failure F_e - Erratic Failure R_c - Recovery Content

R_t - Recovery Timing R_h - Recovery Hang R_e -Recovery Erratic

To confirm the accuracy of the proposed model we compared its quantitative prediction with the real obtained values by taking into account the following metrics: reliability, probability of entering in a failed state and the probability of each failure type.

The reliability results are given in Table 7.6. These results show similar values between the real and the modeled one, bound by a 0.04% error margin. It should also be pointed out that Paravirtualization (PV) presents a higher reliability value than the Hardware Virtual Machine (HVM). This can be explained by the fact that HVM emulates the full-hardware stack incurring in a performance overhead due to the emulation layer. This overhead makes client requests being processed less efficiently, resulting in a higher number of queued requests and a higher likelihood of dropping them, making the system less reliable.

Table 7.6: Reliability

	Real	Modeled	Diff
HVM	97.86%	97.90%	0.04%
PV	99.38%	99.38%	0.0%

The probability of entering in a failed state is defined as an error being since it is activated and can lead to a failure. Table 7.7 shows the reliability figures obtained from actual execution and values that resulted from modeling the system. The results depicted show a difference between both approaches of less than two percent. With regard to the virtualization types, the PV achieves better results since it has a lower probability of entering in a failed state.

Table 7.7: Probability of entering in a failed state

	Real	Modeled	Diff
HVM	14.78%	12.88%	1.9%
PV	9.50%	8.67%	0.83%

The last metric to validate our proposed model is the ‘probability of occurrence' of each failure type. Table 7.8 illustrates the outcomes from the actual experimentation and from our model by considering the two virtualization types, HVM and PV. The results differ from 0.1% to 1.78% between the real and the modeled values, but overall they achieve similar probabilities. As in the previous experiments, the PV virtualization type achieves better results than the HVM.

Table 7.8: Probability of entering in each failure mode

		Real	Modeled	Diff
HVM	Content	3.89%	3.79%	0.10%
	Timing	0.09%	0.09%	0.00%
	Hang	14.29%	12.51%	1.78%
	Erratic	0%	0%	0%
PV	Content	0.72%	0.71%	0.01%
	Timing	0%	0%	0%
	Hang	8.78%	8.07%	0.71%
	Erratic	0%	0%	0%

7.3.2.2 Error propagation validation

In this section we examine the validation of the proposed modeling approach by taking into account error propagation. Figure 7.3 illustrates a Discrete-Time Markov Chain (DTMC) which was simplified to highlight the relevant states and omit the ones that do not have any transition, such as the content failure state (F_c). Our approach models the client performing requests to the VM1. These requests pass through the Dom0 which may propagate an error to other VMs. When this error propagation occurs, it leads to only one type of failure, the hang. Since the occurrence of a hang failure brings the whole system to an inoperative state, we modeled the transition from the halt state from the VMs to the complete failure state. Moreover, the interaction between components Dom0 and VM1 is

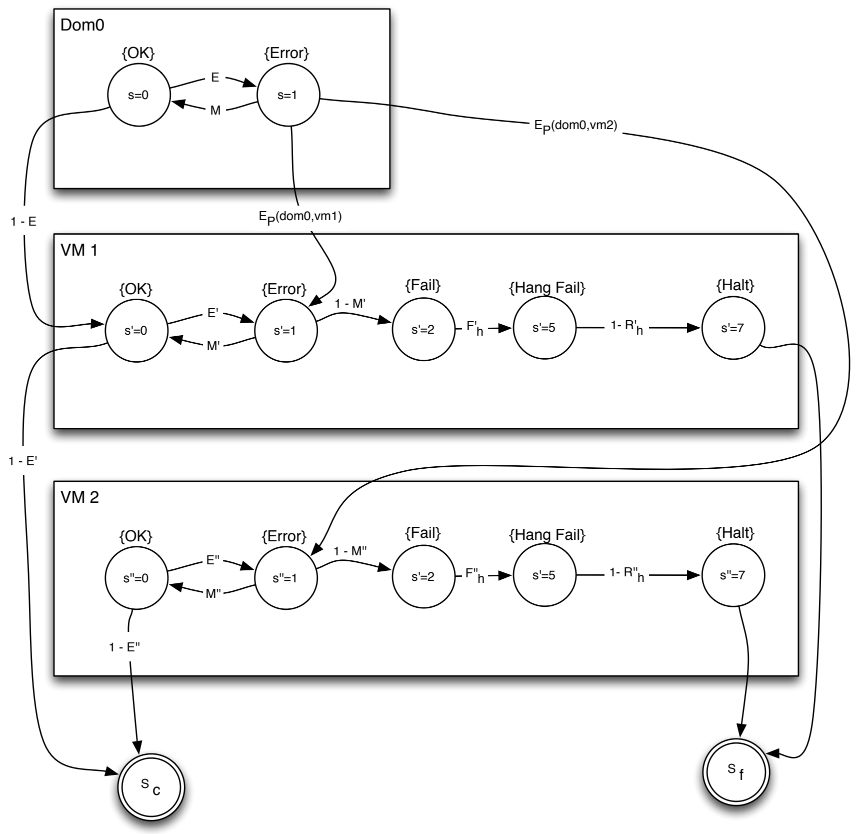


Figure 7.3: DTMC modeling the Dom0 Fault Injection scenario

modeled through a sequential order, which means that Dom0 executes and then passes the control to VM1.

Table 7.9 depicts the failure data collected from Tables 7.3 and 7.4.

The results of this experiment and the estimations of our model are illustrated in Table 7.10. The reliability figures are almost identical in both approaches, with WatchDogD having the higher deviation between real and modeled results, a difference below 4%.

7.3.3 Discussion of the Results

It can be concluded from the failure rate data that were gathered that the Paravirtualization (PV) type achieves better results than the Hardware Virtual Machine (HVM). In addition, the HVM experienced timing failures while PV did not. We

Table 7.9: Modeling Parameters for the Dom0 experiment

	Qemu			Xenwatchdogd		
	VM 1	VM 2	Dom0	VM 1	VM 2	Dom0
M	0.0	0.0	0.93	0.0	0.0	0.76
F_c	0.0	0.0	0.0	0.0	0.0	0.0
F_t	0.0	0.0	0.0	0.0	0.0	0.0
F_h	1.0	1.0	0.0	1.0	1.0	0.0
F_c	0.0	0.0	0.0	0.0	0.0	0.0
R_c	0.0	0.0	0.0	0.0	0.0	0.0
R_t	0.0	0.0	0.0	0.0	0.0	0.0
R_h	0.0	0.0	0.0	0.0	0.0	0.0
R_e	0.0	0.0	0.0	0.0	0.0	0.0
$E_p(\text{dom0, VM1})$	-	-	0.035	-	-	0.12
$E_p(\text{dom0, VM2})$	-	-	0.035	-	-	0.12

Table 7.10: Dom0 Reliability as the probability of non-failure

	Real	Modeled	Diff
Qemu	93.06%	93.51%	0.45%
WatchDogD	76.71%	80.64%	3.93%

assume that this difference is related to the nature of the virtualization type. HVM implements a full emulation of the hardware which results in a slower performance than the Paravirtualization (PV) type, and this explains the occurrence of timing failures in the HVM.

Furthermore, our experimental results show that the isolation layer between

VMs is effective when a fault is injected in one of the VMs. However, when an error is injected in a kernel extension, the error is not only propagated to the guest VMs, but can also cause the system to fail and ultimately, lead it to a completely inoperative state.

With regard to the validity of our approach, both the estimations from our model and the values obtained from the actual Xen system are similar. The difference between both approaches falls in the interval $[0.0\%, 3.9\%]$, where there is, on average, a difference of 0.69% and a median of 0.07%. These figures support the validity our proposed modeling approach.

7.4 Related Work

Surveys on reliability prediction from a software architecture [Gokhale, 2007; Goševa-Popstojanova & Trivedi, 2001; Immonen & Niemelä, 2008] overview the state-of-the-art pointing out venues for future work and current limitations. From the outlined limitations, Gokhale [Gokhale, 2007] reminds the importance of defining component failure models and identifies the area of parameter estimation techniques as being in the infancy at that time (2007). Immonen and Niemelä [Immonen & Niemelä, 2008] point out the lack of support for tools, weak reliability analysis of software components, and weak validation of the methods and their results. The work outlined in this chapter addresses the above limitations by proposing a reliability prediction model that defines component failures models as suggested by Gokhale. In addition, we propose a practical experiment which addresses the lack of parameter estimation techniques and also serves as a validation of our modeling method addressing the limitation raised by Immonen and Niemelä.

Goseva-Popstojanova *et al.* [Goseva-Popstojanova *et al.*, 2005] presented a comparison between different architecture-based reliability prediction methods based on an empirical, large scale and real case-study. Their work reveals that the literature assumes a too simplistic relationship between faults and failures, leading to errors in the analysis. This seminal paper concludes that more work should be conducted by the software testing and reliability research communities in order to explore more realistic relationships between faults and failures. This was a strong motivator for the work of this chapter.

Cortellessa and Grassi [Cortellessa & Grassi, 2007] proposed a method to predict the system reliability by encompassing error propagation. In their work, authors recognize that they faced the same difficulties as other studies, namely the absence of reference values and a lack of parameter estimation techniques to obtain meaningful probabilities of actual internal failures and error propagation.

Filieri *et al.* [Filieri *et al.*, 2010] present a novel approach that deals with multiple failure modes and error propagation among components. Their approach supports the specification of individual components' attitude to produce, propagate, transform and mask different failure modes. Again, as pointed out by the authors in the concluding remarks, they are unable to assess their approach effectiveness due to the lack of real case-studies and actual values of reliability.

The proposed reliability model also accounts for error propagation which consists of an error occurring somewhere in the system. This error can be propagated to other components or to the application output. Hiller *et al.* [Hiller *et al.*, 2001] introduced the concept of error permeability and presented a framework to analyze the propagation and severity of data errors in a software system.

To assess the error propagation between components, Abdelmoez *et al.* [Abdelmoez *et al.*, 2004] proposed an analytical method using fault-injection techniques to estimate the probability of error propagation in a software architecture. Johansson and Suri [Johansson & Suri, 2005] also used fault-injection techniques to assess error propagation in an Operating System (OS). The OS is assumed to be a black-box and it is profiled through a fault injection technique and data error propagation analysis. The results showed that many errors do not propagate or propagate in a robust manner.

In common with some of the above error propagation studies, our approach relies on fault injection techniques to estimate the system failure data. However, it differs in the fact that we use the collected data to study the reliability impact and validate the proposed modeling approach. Thus, our goal is not only the estimation of the failure data of a system, but also provide a realistic, freely-accessible case-study, and actual failure data which can be used by researchers and practitioners to compare and validate different reliability prediction methods.

In short, this paper proposes a novel modeling approach to estimate reliability from a software architecture. It addresses the shortcomings highlighted by research surveys by proposing a more detailed failure behavior including masking, error propagation, multiple failure modes and recoveries. As a result, we validate our approach through a realistic case-study by collecting actual failure data and by asserting the adequacy of the proposed stochastic model.

7.5 Summary

In this chapter we proposed a novel approach to model the failure pathology of a software component. The proposed model extends the already accepted reliability modeling methods, and also addresses their limitations. Moreover, our model encompasses error masking, error propagation and multiple failure modes and thus addresses the modeling shortcomings of reliability prediction methods.

Additionally, we addressed the lack of realistic case-studies available in the research community to compare and validate newly proposed reliability prediction methods. In view of this, we implemented a case-study based on a freely accessible and widely used cloud infrastructure platform, the Xen Hypervisor.

To validate our proposal, we conducted a set of experiments by using this testbed, including a comprehensive series of fault injection runs to emulate the presence of erroneous states in the system when employed to collect its failure rate data. The results show that the figures obtained from our estimation approach are identical with the actual values obtained from the experiment, with an error margin below 4%. Moreover, we have been able to show that a cloud system may propagate errors from the hypervisor to the guest VMs, which will ultimately lead to their complete disruption. As a result, our approach was also able to accurately predict this behavior, and thus validate our error propagation modeling.

Chapter 8

Conclusion

This thesis was prompted by the following research question raised in Chapter 1: **how can reliability be quantitatively assessed and analyzed from a software description while still avoiding time and effort consuming tasks?** Today's architectural evaluation is a manual and expensive task, which is also prone to errors. The verification of quality conformance in software architecture is not an easy process, especially in large-scale software systems where it is practically impossible to manually verify the whole system against the required quality attributes and ensure that are no conflicts between them.

With this in mind, in Chapter 3 we attempted to address one of the shortcomings of today's reliability prediction methods: the lack of automated methods to test the quality of an architecture. The word 'automated' refers to the process of assessing reliability by avoiding the use of manual activities, since it is less error-prone and requires less effort from designers. In light of this, we proposed an approach that automatically generates mathematical models from a software architecture that avoids the need for manual effort and provides correct and error-free formal models. These models enable architects to test, provide and try out different architectural alternatives.

In the same chapter, we described a formal notation to generate stochastic models from software architectures. The aim of this formal notation is that it can be interpreted universally with rigor and unequivocally by other researchers. Our formal notation can be applied in other research works, and extended to other quality attributes as well as reliability, so that it can comprise different ADLs or support different stochastic models.

In Chapter 4 we proposed an automated approach that performs a sensitivity analysis of the system reliability to identify weak architectural elements and interconnections that are performing poorly. In addition, the approach suggests how architectural improvements can be made through a ranking system which orders architectural elements according to their impact on the system reliability. In short, the proposed method provides the means for architects to improve or evolve the designed artifact by performing a thorough systemic analysis without requiring any manual effort or specialized knowledge.

To show the effectiveness of the these methods, in Chapter 5 we validated our automated reliability assessment and analysis methods. The validation was confirmed by comparing our results with those from widely accepted research methods using the same case-studies and the same architectural styles.

In the same chapter, we extended a tool used for designing ADLS as a means of effectively encouraging the adoption of early assessment of reliability from software architectures. We implemented a plugin within an architectural framework tool which was aimed at assisting designers to achieve more quality on their artifacts with a minimum effort.

In Chapter 6 we showed the applicability of our proposed automated assessment and analysis methods to other application domains. This involved applying our method to self-adaptive systems which usually tend to be reactive and trigger adaptations without testing whether they fulfill the desired non-functional goals. In view of this, we applied our automated method of reliability assessment to their reasoning process. The results show that traditional self-adaptive approaches fail to select the best adaptation strategy in unforeseen circumstances, while our approach recovers from the erroneous behavior and improves the ability to reach the desired quality goals. In short, this work reveals the application of our automated methods to different domains and their effectiveness in improving real global issues.

Chapter 7 proposes a new reliability prediction method that aims to include the failure pathology of a software component. Our proposed approach avoids the assumption that when an error occurs it manifests readily as a failure that propagates to the application output. For example, classical reliability assessment methods assumed that when an error occurs in one of the components the whole application fail without taking into account the possibility of it being masked, recovered or even tolerated by the system. Hence, our approach extends classical reliability assessment methods by proposing a method that encompasses error occurrence, error propagation, multiple failure modes and recoveries.

To conclude, the research question that was the driving-force behind this work can now be answered. Reliability can be quantitatively assessed from a software architectural description through the method set out in Chapter 3 and analyzed from the approach adopted in Chapter 4. We avoided time and effort consuming tasks by removing the need for manual activities, as shown in the implementation of the plugin discussed in Chapter 5 and in the application to self-adaptive systems in Chapter 6. Finally, we proposed a new reliability assessment method which in future can be automated by extending the formal notation outlined in Chapter 3.

The recommendations for future work involve applying our automated assessment method to other quality attributes, like performance or maintainability. The goal is to support more non-functional properties from the architecture and assist designers in evaluating trade-offs between different required quality attributes. In addition, we seek to apply our approach to an industrial partner that is interested in developing quality software through the adoption of assessment and analysis techniques from a software architecture. As we progress, new suggestions for future work keep appearing and our goal will only be completed when architects are assisted by assessment and analytical methods that do not involve any manual activities which when combined result in more quality software.

List of Publications

João M. Franco, Raul Barbosa, and Mário Zenha-Rela. **Automated Reliability Prediction from Formal Architectural Descriptions**. In 2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), pages 302–309, Helsinki, Finland, August 2012. IEEE computer society.

João M. Franco, Raul Barbosa, and Mário Zenha-Rela. **Reliability analysis of software architecture evolution**. In Sixth Latin-American Symposium on Dependable Computing (LADC), pages 11–20, Rio de Janeiro, Brazil, April 2013.

João M. Franco. **Self-adaptive system case-study of architecture-based software reliability**. In Latin-American Symposium on Dependable Computing (LADC), Rio de Janeiro, Brazil, April 2013.

João M. Franco, Francisco Correia, Raul Barbosa, and Mário Zenha-Rela. **Affidavit: Automated reliability prediction and analysis of software architectures**. In INForum 2013, pages 54–65. 5th Portuguese Symposium on Informatics, September 2013.

João M. Franco, Raul Barbosa, and Mário Zenha-Rela. **Availability evaluation of software architectures through formal methods**. In QUATIC SEDES, pages 54–65. Conference on the Quality of Information and Communications Technology (QUATIC), September 2014.

Vitor Silva, João M. Franco, Francisco Correia, Raul Barbosa, and Mário Zenha-Rela. **Assessing the performance overhead of a self-adaptive**

system. In INForum 2014, pages 54–65. 6th Portuguese Symposium on Informatics, September 2014.

João M. Franco, Francisco Correia, Raul Barbosa, Bradley Schmerl, Mário Zenha-Rela and David Garlan. **Improving Self-Adaptation Planning through Software Architecture-based Stochastic Modeling**. In Journal of Systems and Software (JSS) 115: 42-60. January 2016.

João M. Franco, Frederico Cerveira, Raul Barbosa, and Mário Zenha-Rela. **Modeling the Failure Pathology of Software Components** In 2016 Joint Conference on the Quality of Software Architectures (QoSA) and Working IEEE/IFIP Conference on Software Architecture (WICSA). Venice, Italy, April 2016.

Appendix A

Generated Prism Files

A.1 Generated Prism file for the Enlist Server strategy

This section sets out the generated Prism code which includes the DTMC for the Discharge Server Strategy illustrated in Figure 6.6.

```
dtmc

// Number of components: 6 + 2 Absorbing states
global s : [1..8] init 1;

const double p_Web0 = 1.0;
const double p_Web2 = 0.9;
const double p_DB = 1.0;
const double p_Web3 = 1.0;
const double p_LB0 = 1.0;
const double p_Web1 = 1.0;

// Component name - LB0
module LB0
    [] s=1 -> p_LB0*0.25:(s'=2) + p_LB0*0.25:(s'=3) +
       p_LB0*0.25:(s'=4) + p_LB0*0.25:(s'=5) +
       (1-p_LB0):(s'=8);
endmodule

// Component name - Web2
module Web2
    [] s=2 -> p_Web2*1.0:(s'=6) + (1-p_Web2):(s'=8);
endmodule
```

```
// Component name - Web3
module Web3
    [] s=3 -> p_Web3*1.0:(s'=6) + (1-p_Web3):(s'=8);
endmodule

// Component name - Web1
module Web1
    [] s=4 -> p_Web1*1.0:(s'=6) + (1-p_Web1):(s'=8);
endmodule

// Component name - Web0
module Web0
    [] s=5 -> p_Web0*1.0:(s'=6) + (1-p_Web0):(s'=8);
endmodule

// Component name - DB
module DB
    [] s=6 -> p_DB:(s'=7) + (1-p_DB):(s'=8);
endmodule

// Absorbing states
module absorbingStates
    // Final states
    [] s=7 -> (s'=7);
    [] s=8 -> (s'=8);
endmodule

label "available" = (s=7);
label "unavailable" = (s=8);
```

A.2 Generated Prism file for the Discharge the Least Reliable

This section sets out the generated Prism code which includes the DTMC for the Discharge Server Strategy illustrated in Figure 6.7.

```
dtmc
// Number of components: 4 + 2 Absorbing states
global s : [1..6] init 1;

const double p_DB = 1.0;
const double p_Web0 = 1.0;
const double p_LB0 = 1.0;
const double p_Web1 = 1.0;

// Component name - LB0
module LB0
    [] s=1 -> p_LB0*0.5:(s'=2) + p_LB0*0.5:(s'=3) +
       (1-p_LB0):(s'=6);
endmodule

// Component name - Web0
module Web0
    [] s=2 -> p_Web0*1.0:(s'=4) + (1-p_Web0):(s'=6);
endmodule
```

```
// Component name - Web1
module Web1
    [] s=3 -> p_Web1*1.0:(s'=4) + (1-p_Web1):(s'=6);
endmodule

// Component name - DB
module DB
    [] s=4 -> p_DB:(s'=5) + (1-p_DB):(s'=6);
endmodule

// Absorbing states
module absorbingStates
    // Final states
    [] s=5 -> (s'=5);
    [] s=6 -> (s'=6);
endmodule

label "available" = (s=5);
label "unavailable" = (s=6);
```

References

Abd-Allah, A. (1997). Extending reliability block diagrams to software architectures. Tech. Rep. Technical Report USC-CSE-97-501, Dept. of Computer Science, Univ. Southern California. 59

Abdelmoez, W., Nassar, D., Shereshevsky, M., Gradetsky, N., Gunnalan, R., Ammar, H., Yu, B. & Mili, A. (2004). Error propagation in software architectures. In *Software Metrics, 2004. Proceedings. 10th International Symposium on*, 384–393. 138

Allen, R. & Garlan, D. (1996). A case study in architectural modelling: The aegis system. In *Proceedings of the Eighth International Workshop on Software Specification and Design (IWSSD-8)*, 6–15, Paderborn, Germany. 38

Avizienis, A., Laprie, J.C., Randell, B. & Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, **1**, 11–33. 17, 120, 122, 123, 126

Baier, C. & Katoen, J.P. (2008). *Principles of Model Checking (Representation and Mind Series)*. The MIT Press. 31, 32

Barais, O., Meur, A.L. & Duchien, L. (2008). *Software Evolution*. Springer Berlin Heidelberg, Berlin, Heidelberg. 72

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. & Warfield, A. (2003). Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, **37**, 164–177. 123

Bass, L., Clements, P. & Kazman, R. (1998). *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 9, 51

Bjørner, D. & Jones, C.B., eds. (1978). *The Vienna Development Method: The Meta-Language*, Springer-Verlag, London, UK, UK. 38

Brosch, F., Buhnova, B., Koziolek, H. & Reussner, R. (2011). Reliability prediction for fault-tolerant software architectures. In *Proceedings of the joint ACM SIGSOFT conference–QoSA and ACM SIGSOFT symposium–ISARCS on Quality of software architectures–QoSA and architecting critical systems–ISARCS*, 75–84, ACM. 21, 58

Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R. & Tamburrelli, G. (2011). Dynamic qos management and optimization in service-based systems. *Software Engineering, IEEE Transactions on*, **37**, 387–409. 116

Calinescu, R., Ghezzi, C., Kwiatkowska, M. & Mirandola, R. (2012). Self-adaptive software needs quantitative verification at runtime. *Communications of the ACM*, **55**, 69. 114

Camara, J. & de Lemos, R. (2012). Evaluation of resilience in self-adaptive systems using probabilistic model-checking. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on*, 53–62. 114

Casanova, P., Schmerl, B., Garlan, D. & Abreu, R. (2011). Architecture-based run-time fault diagnosis. In *Proceedings of the 5th European Conference on Software Architecture*, ECSA'11, 261–277, Springer-Verlag, Berlin, Heidelberg. 42

Cheng, S.W. (2008). *Rainbow: Cost-effective Software Architecture-based Self-adaptation*. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA. 88, 91

Cheng, S.W. & Garlan, D. (2012). Stitch: A language for architecture-based self-adaptation. *J. Syst. Softw.*, **85**, 2860–2875. 92

Cheng, S.W., Garlan, D. & Schmerl, B. (2009). Evaluating the effectiveness of the rainbow self-adaptive system. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '09, 132–141, IEEE Computer Society, Washington, DC, USA. 92

Cheung, L., Roshandel, R., Medvidovic, N. & Golubchik, L. (2008). Early prediction of software component reliability. In *Proceedings of the 30th international conference on Software engineering*, 111–120, ACM, New York, New York, USA. 60

Cheung, R. (1980). A user-oriented software reliability model. *IEEE Transactions on Software Engineering*, **6**, 118–125. 7, 21, 24, 37, 42, 44, 58, 83, 84, 119, 123

Clarke, E.M. & Emerson, E.A. (1981). Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, ed., *Proceedings of the Workshop on Logic of Programs*, LNCS 131, 52–71, Springer-Verlag. 25

Cortellessa, V. (2002). Early reliability assessment of UML based software models. *Electrical Engineering*, 302–309. 21, 58

Cortellessa, V. & Grassi, V. (2007). A modeling approach to analyze the impact of error propagation on reliability of component-based systems. In *Component-Based Software Engineering*, vol. 4608 of *Lecture Notes in Computer Science*, 140–156, Springer Berlin Heidelberg. 138

De Alfaro, L. (1998). *Formal verification of probabilistic systems*. Ph.D. thesis, Stanford, CA, USA, aAI9837082. 31

Emerson, E.A. (2008). The beginning of model checking: A personal perspective. In *SPIN*, 27–45. 25, 32

Epifani, I., Ghezzi, C., Mirandola, R. & Tamburrelli, G. (2009). Model evolution by run-time parameter adaptation. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, 111–121, IEEE Computer Society, Washington, DC, USA. 116

European Space Agency (ESA) (1988). Software reliability modeling study. Invitation to tender AO/l-2039/87/NL/IW. 21

Feiler, P.H., Gluch, D.P. & Hudak, J.J. (2006). The architecture analysis & design language (AADL): An introduction. Tech. rep., Software Engineering Institute. 15, 38

Filieri, A., Ghezzi, C., Grassi, V. & Mirandola, R. (2010). Reliability analysis of component-based systems with multiple failure modes. In *Component-Based Software Engineering, 13th International Symposium, CBSE 2010, Prague, Czech Republic, June 23-25, 2010. Proceedings*, 1–20. 119, 138

Filieri, A., Ghezzi, C., Leva, A. & Maggio, M. (2011). Self-adaptive software meets control theory: A preliminary approach supporting reliability requirements.

2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), **0**, 283–292. 115, 123

Franco, J.M., Barbosa, R. & Zenha-Rela, M. (2012). Automated reliability prediction from formal architectural descriptions. In *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, 302–309, IEEE computer society, Helsinki, Finland. 75

Franco, J.M., Barbosa, R. & Zenha-Rela, M. (2013). Reliability analysis of software architecture evolution. In *Sixth Latin-American Symposium on Dependable Computing (LADC)*, 11–20, Rio de Janeiro, Brazil. 75

Fredericks, E.M., Ramirez, A.J. & Cheng, B.H.C. (2013). Towards run-time testing of dynamic adaptive systems. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '13, 169–174, IEEE Press, Piscataway, NJ, USA. 88

Gallotti, S., Ghezzi, C., Mirandola, R. & Tamburrelli, G. (2008). Quality prediction of service compositions through probabilistic model checking. In *Proceedings of the 4th International Conference on Quality of Software-Architectures: Models and Architectures*, QoSA '08, 119–134, Springer-Verlag, Berlin, Heidelberg. 114

Garlan, D. & Shaw, M. (1994). An Introduction to Software Architecture. *Knowledge Creation Diffusion Utilization*. 9, 10, 51, 59

Garlan, D., Monroe, R. & Wile, D. (1997). Acme: An architecture description interchange language. In *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '97, 7–, IBM Press, Toronto, Ontario, Canada. 15, 38, 56, 74, 75

Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B. & Steenkiste, P. (2004). Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, **37**, 46–54. 92

Garlan, D., Bachmann, F., Ivers, J., Stafford, J., Bass, L., Clements, P. & Merson, P. (2010). *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2nd edn. 8

Gokhale, S. (2006). Analytical models for architecture-based software reliability prediction: A unification framework. *Reliability, IEEE Transactions on*, **55**, 578–590. 120

GOKHALE, S.S. (2007). Architecture-Based Software Reliability Analysis : Overview and Limitations. *IEEE Transactions On Dependable And Secure Computing*, **4**, 32–40. 21, 42, 56, 58, 119, 137

GOKHALE, S.S. & TRIVEDI, K.S. (2002). Reliability Prediction and Sensitivity Analysis Based on Software Architecture. *Reliability Engineering*. 24, 44, 58, 61, 72, 83, 84

GOSEVA-POPSTOJANOVA, K., HAMILL, M. & PERUGUPALLI, R. (2005). Large Empirical Case Study of Architecture-Based Software Reliability. In *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, 43–52, IEEE. 24, 58, 73, 137

GOŠEVA-POPSTOJANOVA, K. & TRIVEDI, K. (2001). Architecture-based approach to reliability assessment of software systems. *Performance Evaluation*, **45**, 179–204. 7, 22, 23, 42, 58, 119, 121, 123, 137

GRINSTEAD, CHARLES M. AND SNELL, L.J. (2006). *Grinstead and Snell's Introduction to Probability*. July, American Mathematical Society, version da edn. 58

HILLER, M., JHUMKA, A. & SURI, N. (2001). An approach for analysing the propagation of data errors in software. *Proceedings International Conference on Dependable Systems and Networks*, 161–170. 138

HINTON, A., KWIATKOWSKA, M., NORMAN, G. & PARKER, D. (2006). Prism: A tool for automatic verification of probabilistic systems. In *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06), volume 3920 of LNCS*, 441–444, Springer. 33

HOARE, C.A.R. (1978). Communicating sequential processes. *Commun. ACM*, **21**, 666–677. 38

IBM CORP. (2004). *An architectural blueprint for autonomic computing*. IBM Corp., USA. 89, 113

IMMONEN, A. & NIEMELÄ, E. (2008). Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and Systems Modeling*, **7**, 49–65. 22, 56, 58, 73, 137

ISO/IEC/IEEE (2011). ISO/IEC/IEEE systems and software engineering – architecture description. *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, 1–46. 9, 14, 39, 56, 72, 114

Issarny, V. & Zarras, A. (2003). Software architecture and dependability. In M. Bernardo & P. Inverardi, eds., *SFM*, vol. 2804 of *Lecture Notes in Computer Science*, 259–286, Springer. 14

Jeannet, B., D'Argenio, P. & Larsen, K. (2002). Rapture: A tool for verifying Markov decision processes. In I. Cerna, ed., *Proc. Tools Day, affiliated to 13th Int. Conf. Concurrency Theory (CONCUR'02)*, Technical Report FIMU-RS-2002-05, Faculty of Informatics, Masaryk University, 84–98. 33

Johansson, A. & Suri, N. (2005). Error propagation profiling of operating systems. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, 86–95. 138

Katoen, J.P., Zapreev, I.S., Hahn, E.M., Hermanns, H. & Jansen, D.N. (2009). The Ins and Outs of The Probabilistic Model Checker MRMC. In *Quantitative Evaluation of Systems (QEST)*, 167–176, IEEE Computer Society, `www.mrmc-tool.org`. 33

Kirwan, B. (1994). *A Guide to Practical Human Reliability Assessment*. London: Taylor & Francis Ltd. 107

Kwiatkowska, M., Norman, G. & Parker, D. (2007). Stochastic model checking. In M. Bernardo & J. Hillston, eds., *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07)*, vol. 4486 of *LNCS (Tutorial Volume)*, 220–270, Springer. 26, 27, 28, 122

Kwiatkowska, M., Norman, G. & Parker, D. (2009). Prism: Probabilistic model checking for performance and reliability analysis. *ACM SIGMETRICS Performance Evaluation Review*, **36**, 40–45. 56, 114

Kwiatkowska, M., Norman, G. & Parker, D. (2011). PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan & S. Qadeer, eds., *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, vol. 6806 of *LNCS*, 585–591, Springer. 56

Laprie, J. & Kanoun, K. (1996). Software reliability and system reliability. *Handbook of software reliability engineering*, 27–69. 24

Lemos, R., Giese, H., Muller, H.A., Shaw, M., Andersson, J., Baresi, L., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Desmarais, R., Dustdar, S., Engels, G., Geihs, K., M. Goeschka, K., Gorla, A., Grassi, V., Inverardi, P., Karsai, G., Kramer, J., Litoiu, M., Lopes, A., Magee, J., Malek, S., Mankovskii, S., Mirandola, R., Mylopoulos, J., Nierstrasz, O., Pezze, M., Prehofe, C., Schäfer, W., Schlichting, R., Schmerl, B., B. Smith, D., P. Sousa, J., Tamura, G., Tahvildari, L., M. Villegas, N., Vogel, T., Weyns, D., Wong, K. & Wuttke, J. (2013). Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In R. de Lemos, H. Giese, H. Müller & M. Shaw, eds., *Software Engineering for Self-Adaptive Systems*, vol. 7475 of *Dagstuhl Seminar Proceedings*, 1–26, Springer. 113

Littlewood, B. (1979). Software Reliability Model for Modular Program Structure. *IEEE Transactions on Reliability*, **R-28**, 241–246. 24

Lo, J.H., Huang, C.Y., Chen, I.Y., Kuo, S.Y. & Lyu, M.R. (2005). Reliability assessment and sensitivity analysis of software reliability growth modeling based on software module structure. *Journal of Systems and Software*, **76**, 3–13. 44, 58, 61, 72, 83, 84

Luckham, D.C., Kenney, J.J., Augustin, L.M., Vera, J., Bryan, D. & Mann, W. (1995). Specification and analysis of system architecture using rapide. *IEEE Trans. Softw. Eng.*, **21**, 336–355. 15

Lyu, M.R., ed. (1996). *Handbook of software reliability engineering*. McGraw-Hill, Inc., Hightstown, NJ, USA. 17, 20

Macías-Escrivá, F.D., Haber, R., del Toro, R. & Hernandez, V. (2013). Self-adaptive systems: A survey of current approaches, research challenges and applications. *Expert Systems with Applications*, **40**, 7267–7279. 88

Martens, A., Koziolek, H., Becker, S. & Reussner, R. (2010). Automatically Improve Software Architecture Models for Performance , Reliability , and Cost Using Evolutionary Algorithms. *Population (English Edition)*. 56, 59

Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L. & Wolf, E.L.

(1999). An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, **14**, 54–62. 89

Pengoria, D., Kumar, S. & Se, M.S. (2009). A Study on Software Reliability Engineering Present Paradigms and its Future Considerations. *Computing*. 22, 56, 58

Perry, D.E. & Wolf, A.L. (1992). Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, **17**, 40–52. 8

Potter, B., Till, D. & Sinclair, J. (1996). *An Introduction to Formal Specification and Z*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edn. 38, 39

Qian, L., Luo, Z., Du, Y. & Guo, L. (2009). Cloud computing: An overview. In M. Jaatun, G. Zhao & C. Rong, eds., *Cloud Computing*, vol. 5931 of *Lecture Notes in Computer Science*, 626–631, Springer Berlin Heidelberg. 123

Reussner, R. & Heinz W. (2003). Reliability prediction for component-based software architectures. *Journal of Systems and Software*, **66**, 241–252. 21, 24, 58, 119

Salehie, M. & Tahvildari, L. (2009). Self-adaptive software. *ACM Transactions on Autonomous and Adaptive Systems*, **4**, 1–42. 88, 113

Schmerl, B. & Garlan, D. (2004). AcmeStudio: supporting style-centered architecture development. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, 704 – 705. 74

Shooman, M.L. (1976). Structural models for software reliability prediction. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, 268–280, IEEE Computer Society Press, Los Alamitos, CA, USA. 22

Sommerville, I. (2000). Software engineering, 6th Edition (Slides). *Software Engineering, IEEE Transactions on*. 9

Spivey, J.M. (1989). *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. 38

Storey, N.R. (1996). *Safety Critical Computer Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 20, 95

Sun, J., Liu, Y. & Dong, J.S. (2008). Model checking csp revisited: Introducing a process analysis toolkit. In *In ISoLA 2008*, 307–322, Springer. 34

Taylor, R.N., Medvidovic, N. & Dashofy, E.M. (2009). *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing. 8, 10, 15, 17, 21, 38, 56

Tijms, H.C. (2003). *A First Course in Stochastic Models*. Wiley. 26, 32

Villegas, N.M., Müller, H.a., Tamura, G., Duchien, L. & Casallas, R. (2011). A framework for evaluating quality-driven self-adaptive software systems. *Proceeding of the 6th international symposium on Software engineering for adaptive and self-managing systems - SEAMS '11*, **1**, 80. 91, 92

Wang, W.l., Pan, D. & Chen, M.H. (2006). Architecture-based software reliability modeling. *Journal of Systems and Software*, **79**, 132–146. 55, 59, 85, 119, 123

Yacoub, S., Cukic, B. & Ammar, H. (2000). Scenario-based reliability analysis of component-based software. *Proceedings 10th International Symposium on Software Reliability Engineering (Cat. No.PR00443)*, 22–31. 21, 58

Zheng, T., Woodside, M. & Litoiu, M. (2008). Performance model estimation and tracking using optimal filters. *Software Engineering, IEEE Transactions on*, **34**, 391–406. 115