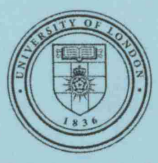


UNIVERSITY OF LONDON THESIS

Degree PhD Year 2008 Name of Author NUNES RODRIGUES, Gemaina

A MODEL DRIVEN APPROACH FOR SOFTWARE RELIABILITY PREDICTION

Genaína Nunes Rodrigues

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of the

University of London.

Department of Computer Science

University College London

March 2008

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

I, Genaína Nunes Rodrigues, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Aos meus pais,
pela minha vida
e pelo amor incondicional.

Abstract

Software reliability, one of the major software quality attributes, quantitatively expresses the continuity of correct service delivery. In current practice, reliability models are typically measurement-based models, and mostly employed in isolation at the later stage of the software development process, after architectural decisions have been made that cannot easily be reversed; early software reliability prediction models are often insufficiently formal to be analyzable and not usually connected to the target system.

We postulate it is possible to overcome these issues by supporting software reliability engineering from requirements to deployment using scenario specifications. We contribute a novel reliability prediction technique that takes into account the component structure exhibited in the scenarios and the concurrent nature of component-based systems by extending scenario specifications to model (1) the probability of component failure, and (2) scenario transition probabilities. Those scenarios are subsequently transformed into enhanced behaviour models to compute the system reliability.

Additionally we enable the integration between reliability and development models through profiles that extend the core Unified Modelling Language (UML). By means of a reliability profile, the architecture of a component-based system can express both method invocations and deployment relationships between the application components in one environment.

To facilitate reliability prediction, and determine the impact of concurrency on systems reliability, we have extended the Label Transition System Analyser Tool (LTSA), implementing a plugin for reliability analysis.

Finally, we evaluate our analysis technique with a case study focusing on Condor, a distributed job scheduler and resource management system. The purpose of the case study is to evaluate the efficacy of our analysis technique and to compare it with other reliability techniques.

Acknowledgements

First, I would like to express my most profound gratitude to David Rosenblum. There are no words to express how important your guidance, your patience and your unbounded help were, David. My eternal admiration and gratitude for all I received from you!

To the UCL academics: Wolfgang Emmerich, Anthony Finkelstein (my very first contact from UCL by email), Graham Roberts, Angela Sasse and Søren-Aksel Sørensen, Steve Hailes and Cecilia. My special thanks to Wolfgang for giving me the opportunity to embrace this challenge and for guiding me to strive for the best! A great thank you to Graham for guiding me through the first 15 months of my PhD! To Sebastian Uchitel for bringing up inspiring questions and providing guidance in our meetings at Imperial College. Thanks a ton, Seb! To Jonas Wolf, Robert Chatley and Chuan, thanks for the support in hacking LTSA ;-)

To Dr. Masaharu Taniguchi for this tremendous Seicho-no-Ie teaching, the rock I can stand upon! To all my Seicho-no-Ie friends in UK and in Brazil! To José and Madalena Pinto, my parents in London. I am definitely a better person thanks to you! To Priscilla, Crezo, José Roberto and Suelma, my family from Goiânia who will always be in my heart! A special thank you to Priscilla for the incredible and unconditional support. My sister Pri, love you tons!

To my dearest sister Gilva and her family: Gabriel, Jéssica and Stephanie for always sending good vibes, even from far away. A titia ama vocês! To the Wichrowski family (Solimar, Mariana, Gabriel, Victor and Caio) for supporting me in another important chapter of my life! To my friend Paulo Baptista for showing that friendship is not bound to distance!

For my UCL friends, I can not dare to list them all, but my most special thanks to Stef and Soc (and the Greek family I also joined), Lê, Carina, Rami (my brother!), Licia (the benchmark!), Franco (grazie per tutto, amico!), James (the intellectual inspiration ;-), Clovis (for the lengthy Condor discussions), Phil Cook, Vladimir, Costin, Miro, Vito, Panu, Andy Dingwall, Mirco, Bojenka, Andy Maule, Chris Wallenta (great games!), Luís Álvaro, Mo, Rae, Elisa and Ettore, Michele, Dimitris, Sonia Sha, the newcomers Valentina, Afra, Bence and Arun.

Last but not least to my parents, to whom I am eternally grateful. My light in dark days and my fortress in stormy days. All I am and I have achieved would be nothing without you!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software reliability is one of the major software quality attributes, which quantitatively expresses the continuity of correct service delivery. Reliability models are typically measurement-based models, and mostly employed in isolation at the later stage of the software development process. In current practice, early software reliability prediction models are often insufficiently formal to be analyzable and not usually connected to the target system. Additionally, despite the vast work that has been done in software reliability, much work is still needed, especially in the component-based development arena regarding availability of software component reliability information following a clear failure classification scheme. Aiming at addressing those problems, we contribute a novel reliability prediction technique that leverages reliability analysis in early stages of software development by taking into account (1) the component structure exhibited in the scenarios elicited in the requirements phase and (2) the concurrent nature of component-based systems. Following that contribution, we propose a means to accomplish reliability design and analysis for model driven engineering following the Model Driven Architecture standards. By doing that, we contribute to the task of systematically integrating reliability modelling from the early to the late stages of software engineering and thus semantically integrating analysis, design and deployment models for reliability into one environment.

1.1 Historical Background

Component-based development (CBD) has become a pervasive practice in software engineering development. CBD Architectures (CBDAs) support distributed execution across machines running on different operating systems platforms. Additionally, CBDAs rely on the construction and deployment of software systems that have been assembled from components [Emm02]. One of the advantages of applying a component-based approach is reusability. It is easier to integrate classes into coarse-grained units that provide one or more clearly defined interfaces. However, the lack of interoperability among diverse CBDAs may be one of the major problems that hinders the adoption of distributed component technologies. Once a platform has been chosen and the system has been developed, porting to a different platform becomes troublesome.

To fill the gap, the OMG has focused on paving the way to provide CBDAs interoperability standards through the Model Driven Architecture (MDA). Essentially, "the MDA defines an approach to IT system specification that separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform" [OMG01]. To accomplish this approach, MDA structures the system into key models: the Platform Independent Models (PIMs) and the Platform Specific Models (PSMs). While a PIM provides a formal specification of the structure and function of the system that abstracts away technical details, a PSM expresses that specification in terms of the model of the target platform. Basically, PIMs are mapped to PSMs when the desired level of refinement of PIMs is achieved.

The Unified Modeling Language (UML) is part of the core representation of those models. According to the OMG, UML supports the formalization of abstract, though precise, models of the state of an object, with functions and parameters provided through a predefined interface [OMG01]. Furthermore, UML models facilitate the assessment of a design in the early stages of software development, when it is easier and cheaper to make changes. As a result, mechanisms to represent various aspects of system design and analysis can be expressed within one consistent language for specifying, visualising, constructing and documenting the artifacts of software systems. In order to express all aspects and concerns of the software development lifecycle, UML extension mechanisms, i.e. profiles, have been used to introduce capabilities for representing non-functional concerns in UML models [OMG04b, OMG05]. UML profiles contain mechanisms that allow the extension of metaclasses for different purposes, tailoring the UML metamodel for different platforms (e.g. J2EE or .NET) or domains (e.g. real-time or business process) [OMG04a]. The profile mechanism of UML provides lightweight extension mechanism to the UML standard by allowing one to define sets of stereotypes, tagged values

and Object Constraint Language (OCL) rules. In the notation of current UML 2 specification, stereotypes are metaclasses, tagged values are standard metaattributes, and profiles are specific kinds of packages. The OCL rules in the UML profiles are used in order to enforce well-formedness rules that constrains, but are still consistent with, those specified in the reference metamodel.

It is well known the great interest from the software engineering community in closing the gap between commercial design tools and quantitative evaluation of software systems. Many are the reasons behind this fact, especially those related to software quality. Software has become an ubiquitous element of our daily lives and the literature points out that its size has been growing exponentially over the past 40 years [Lyu07]. Among software quality attributes, software reliability is one of the major ones as it can quantitatively express the continuity of correct service delivery. Included in the concept of reliability there are metrics for reliability and means to deliver correct service in the presence of faults from software and hardware. The mean time to failure (MTTF) and availability are among the most common metrics for reliability. As a way to improve reliability there is fault tolerance, which works by preserving the delivery of correct service in the presence of active faults [ALR01]. Usually, fault tolerance is implemented by error detection followed by fault handling and system recovery. Current CBDAs, such as Enterprise Java Beans and CORBA, support a considerable range of fault tolerance mechanisms in order to provide reliable services [Emm00a], such as replication and failure transparency.

The greater the demand on integrating design and analysis, the greater the need to provide software engineers, researchers and practitioners with ability to manage software quality throughout its lifecycle; in particular, during its early stages, for the well known reasons of saving costs, time and efforts. This is the rationale behind a great deal of interest in model driven engineering. As a result, providing means to increase the ability to manage software quality has been one of the key focus of the MDA community. The UML 2.0 Specification itself has augmented the previous UML version so that software system characteristics and the dynamic aspects of software behaviour can be represented more accurately [OMG04a]. On top of UML 2 new features, OMG has also specified the UML Profile for Modeling Quality of Service and Fault-Tolerance (QoS Profile shortly). The QoS Profile defines a set of UML extensions to represent dependability attributes, where reliability is one of them[1], and other non-functional requirements using the lightweight extension mechanisms of UML [OMG04b].

[1]In accordance with Avizienis et al. [ALR01] definition of dependability, which is the ability of a computing system to deliver service that can justifiably be trusted. Among attributes of dependability are: reliability, safety, confidentiality, integrity and maintainability.

1.2 The Problem Definition

In spite of those interests and efforts, techniques available to validate a design against non-functional properties, particularly reliability, often require significant expertise unrelated to the usual business of engineering software. As reliability measures quantitatively the quality of correct service delivery, it is probably the most important characteristic for the software engineering discipline. However, as pointed out by Michael R. Lyu [Lyu07], Software Reliability Engineering (SRE) is not yet fully delivering its promise. Major reasons behind it is that reliability models are typically measurement-based models, and therefore employed in isolation at the later stage of the software development process. On the other hand, early software reliability prediction models are often insufficiently formal to be analyzable and not usually connected to the target system. Therefore, as mentioned by Lyu [Lyu07]:

> There is currently a need for a creditable end-to-end software reliability model that can be directly linked to reliability prediction from the very beginning (i.e. software design), so as to establish a systematic SRE procedure that can be certified, generalized and refined.

In the pursue of modeling software reliability of component-based development, there are some key questions that need to be addressed during the design of the system architecture, such as:

- Identifying the reliability each component needs to have in order to meet the intended system reliability. Likewise, identifying components that more significantly impact on the system reliability. Usually, software analysts have to exploit complex analytical functions describing the system behaviour in order to find out how sensitive the system reliability is to the components' failure probability.

- Requirement: stakeholders want to know the effect that different system features and execution scenarios have on system reliability. Therefore, it may be desirable to know if or how different designs affect the system reliability before the decisions are made and costs committed.

- If one wants to investigate the component-based software reliability from designs, it is also plausible to investigate through software analysis if the concurrent nature of the software will reveal intended or unintended component interactions and if those interactions will affect the system reliability.

However, means to answer those questions are few and inadequate. The vast majority of techniques for reliability prediction are tailored to be used during the late phase and not during the requirements elicitation or design phases of the software lifecycle.

Although MDA promises to overcome important unsolved problems in software engineering, it has not specified ways to comprehensively represent software reliability yet. In fact, OMG specified reliability in the QoS Profile as a *QoSCharacteristic*, which *QoSDimension* provides support for quantification of reliability in terms of the ability of a system to keep operating correctly over a period of time. However, we believe those properties are not comprehensive enough to support reliability engineering, in particular analysis, as it does not address the modeling of dynamic aspects of a system, such as scenarios, component interactions or information regarding transitions between states of a modeled system, often required in modeling and analyzing component-based software systems. As a system consists of a set of interacting components such that the interactions can reveal faults [ALR01], modeling and annotating those interactions appropriately can assist us in predicting software reliability. On the other hand, dynamic aspects have been defined in the UML Profile for Schedulability, Performance and Time Specification [OMG05] (henceforth referred to as the SPT Profile), but they were not incorporated into the QoS Profile. In fact, this gap may result from lack of adequate techniques to predict reliability using simple and well known constructions and modeling elements, e.g. Message Sequence Charts (MSCs), to describe system requirements.

Therefore, as a result of that gap, model-driven software development can lead to undesirable situations during the software executions.

1.3 Contributions

We postulate it is possible to overcome those issues by providing means to support software reliability engineering from requirements to deployment level, with adequate analysis, through appropriate mappings. An approach to obtain an early estimate of the system reliability is to use scenario specifications which are subsequently transformed into enhanced behaviour models.

Concentrating on the levels of our process with an adequate and systematic approach, the desired reliability of software systems will be reported early on software development phases according to the required reliability property defined in the architecture level. Based on that process, our thesis then realizes the following contributions:

1.3.1 Reliability Prediction by Model Synthesis From Scenarios

This contribution consists of a method to predict software system reliability from scenarios specifications [RRU05c]. The approach involves extending a scenario specification to model

(1) *the probability of component failure*, and (2) *scenario transition probabilities* derived from an operational profile of the system. The contribution of this approach is a reliability prediction technique that takes into account the component structure exhibited in the scenarios and the concurrent nature of component-based systems.

The approach to verify the reliability property for the software system, i.e. the analysis model, starts with a set of scenarios and a high-level message sequence chart (HMSC). The HMSC is annotated with *scenario transition probabilities* derived from an operational profile of the system [Mus93], which accounts for the relative frequency with which system usage results in a transition from one scenario to another. We synthesise from the scenarios a deterministic probabilitistic behaviour model for each system component. Each component model is then extended to model the probabilistic occurrence of component *failure*. The resulting probabilistic component models are composed in parallel and used to predict the overall reliability of the component-based system according to Cheung's user-oriented reliability model [Che80].

The analysis also comprises sensitivity analysis to identify components and usage profiles with greater impact on the system reliability [RRU05b]. Through the sensitivity analysis we can find out how the system reliability is sensitive to the (1) *components reliabilities* and (2) *scenario transition probabilities*. These two analyses can help us to identify components and scenarios transitions that could represent a threat to the reliability of the software system. Taking into account the concurrent nature of component-based software systems, we are also able to analyse what effects the prevention of undesirable implied scenarios cause to our reliability prediction's.

1.3.2 Conforming Reliability Modeling to a Standard

The target of this contribution is a solution for model driven reliability engineering following the principles of the MDA standard [RRE04, RRU05a]. This contribution focuses particularly on addressing reliability modeling and analysis into MDA. The syntax and semantics of MDA models are represented through profiles that extend the core UML using metamodeling techniques. By this means, modeling reliability can be considerably facilitated as complexity in making software more reliable is raised to a higher level of abstraction. To achieve this goal, this contribution relies on reference models specifications such as [OMG05] as well as extensions of the UML metamodels.

By means of a reliability profile, the architecture of an application can express both method invocations and deployment relationships between the application components. All in all, the reliability profile comprises three other major profiles: the design, the analysis and the deployment. In the design profile, meta-modeling techniques are used to map out the reliability

property and fault tolerance mechanism in a profile. As replication is a common practice to achieve fault tolerance, we choose to represent it as a fault-tolerance model comprised in the reliability profile we propose. In the analysis profile, the reliability property for the system is verified and a set of rules for the mapping between design and analysis is specified. Finally, in the deployment profile, components are modeled in a distributed-wise configuration according to the target reliability for the system. These three profiles mainly extends two specifications: (1) the UML Profile for Schedulability, Performance and Real-Time Specification(SPT Profile) [OMG05] and (2) the UML Specification [OMG04a].

1.3.3 Analysis Framework Implementation

To facilitate reliability prediction analysis and elicit impact of concurrency for systems reliability we have extended the Label Transition System Analyser Tool (LTSA) [UCKM03], by implementing a plugin for reliability analysis [RRW07]. LTSA is a tool that allows using behaviour models of distributed systems as prototypes for exploring system behaviour, and for automated checking of model compliance to properties (i.e. , model checking). To support reliability prediction, we annotate a scenario specification with probability annotations and use LTSA to process the annotated scenarios. By scenarios we mean as partial descriptions of how components interact to provide system functionality. A scenario specification is formed by composing multiple scenarios possibly from different stakeholders.

Benefits of using LTSA for reliability analysis is twofold. The first is that LTSA enables the system stakeholder to model the system closer to their perspective instead of requiring the users to provide very fine grained state machine models of the system. Another benefit of using LTSA is the ability of identifying interactions that arise from the concurrent nature of system's behaviour, the so-called implied scenarios [UKM04], which potentially incur in affecting the system reliability. These two benefits together make LTSA a suitable tool for reliability analysis. However, the purpose of the tool is not to model all the component interactions comprising a system. The aim is to provide at a certain level of abstraction what one can expect of the system with regard to its reliability and identify components and scenario transitions with higher impact on a system's reliability by means of sensitivity analysis.

In order to tie our analysis method to our UML profiles, we have integrated our UML profile for reliability with analysis by transforming UML behaviour models into LTSA. The transformation consists of (1) parsing the XML Metadata Interchange (XMI) form of the UML model, which is the standard representation of UML models in XML [OMG02b], and (2) generating the XML input format accepted by LTSA. We implemented the transformation of our UML profile to LTSA in XSLT [W3C99]. XSLT describes rules for transforming a source doc-

ument in a tree format (such as an XML file) into a result document described also by a tree. It therefore suits our need to transform the XMI representation of a UML model into the XML format accepted by LTSA. The transformation process is rather straightforward though.

1.3.4 Analysis Technique Evaluation

The aim to conduct a significant case study is to compare the analysis results we obtain by applying our analysis technique on a real life case study. Although our technique has the character of prediction, we evaluate, from the quantitative aspect, the accuracy of the results produced from our technique compared to other reliability prediction models described in the literature.

We use Condor to evaluate our reliability analysis technique. Condor is a distributed job scheduler and resource management system, as described in [CWT+04]. Condor provides means for users to submit jobs as executable programs. Additionally, based on a set of parameters (e.g. job requirements, resource ownership and workload policies) Condor manages the execution of those jobs on resources selected out of a pool. We apply our technique on Condor in order to predict its reliability and to compare our results with other reliability techniques we present on Chapter 6. In addition to evaluating our technique when modelling systems like Condor, which has a high workload of tasks, we also want to verify the feasibility as well as the effort required in obtaining the input values to apply our analysis technique.

1.4 Thesis Outline

This document is structured as follows: In Chapter 2 we provide the background information for the work we propose on this thesis. In Chapter 3 we we introduce an example to illustrate the class of application our reliability technique aims at and the motivation behind our work. In Chapter 4 we describe a method to predict software system reliability as a function of component reliability estimates. We annotate a scenario specification with probabilistic properties and use a probabilistic labelled transition system (LTS) synthesised from the scenario specification for the software reliability prediction. In Chapter 5 we present our solution for model driven reliability engineering following the principles of the MDA standard. We model three different profiles: one for the reliability conceptual model, one for fault tolerance, one for fault tolerance on EJB and one for reliability analysis based on scenarios. In Chapter 6, we present a case study we analyse our novel technique for reliability prediction. We finally conclude in Chapter 7 where we summarize our contributions on this thesis and explore directions for future work.

Chapter 2

Background Literature

Software reliability can be defined according to two major concepts: one that says that reliability is simply the continuity of correct service [ALR01], and the other which says that it is the probability of failure-free operation of a computer program for a specified period of time in a specified environment [MIO87]. In this thesis we follow the first and more generic definition of reliability, irrespective to time. Our purpose is to provide a systematic approach to support software reliability engineering from requirements to deployment. In this chapter, we provide background information underlying our work, which comprises three major fields: firstly about the standard MDA process, secondly on the field of software reliability engineering and thirdly on model synthesis from scenario specifications. There has been much work in those areas, especially the first two and many others coming out. We present in this chapter the basis in which this thesis is grounded.

2.1 A Short Introduction to the MDA

Nowadays, the demand for distributed systems is increasing as they promote the integration of legacy components and the design of non-functional properties. However, the construction of distributed systems would be far more complex than building a client-server system if it were not for the use of middleware. In order to simplify the construction of distributed systems, middleware has had an important influence on the software engineering research agenda.

The choice of which middleware a software system is based on is mostly influenced by non-functional requirements. However, the support for non-functional requirements usually differ from one platform to another. For example, components developed for Enterprise Java Beans are quite different from components for CORBA or .NET. Once a software system is built based on a particular middleware, the cost of future change to a different underlying platform can be extremely high [Emm00b]. Therefore, the construction of distributed systems using middleware requires principles, notations, methods and tools compatible with capabilities provided by current middleware products. And more importantly, as the underlying infrastructure shifts over time and middleware continues to evolve, the designed business logic of a system should be kept stable.

To address these problems, the OMG has developed a set of specifications that are referred to as the Model Driven Architecture [Emm02]. Essentially, "the MDA defines an approach to IT system specification that separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform" [OMG01]. The Platform Independent Models (PIMs) and the Platform Specific Models (PSMs) are the basic structure where the MDA approach relies on. While a PIM provides a formal specification of the structure and function of the system that abstracts away technical details, a PSM expresses that specification in terms of the model of the target platform. PIMs are mapped to PSMs when the desired level of refinement of PIMs is achieved. The ability to transform a higher level PIM into a PSM raises the level of abstraction at which a developer can work. This ability allows a developer to cope with more complex systems with less effort.

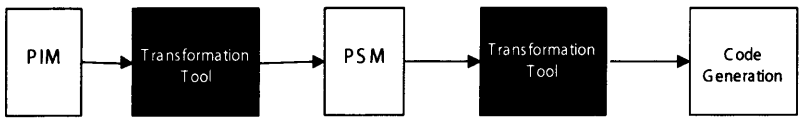


Figure 2.1: Major Steps in the MDA Development Process [KWB03]

Although the MDA software development lifecycle may seem to resemble the same of the

traditional one, there is a crucial difference. Traditionally, tools generate some code from a model, but such generation is not complete enough to prevent handmade work. MDA transformation, on the other hand, are intended to be executed by tools as shown in Figure 2.1. In fact, the transformation step from the PSM to the code generation is not new. The innovation and benefits of MDA relies on the automatic transformation from PIM to PSM. This is where time and effort can be saved when, for example, software analysts and software engineers have to decide on which underlying platform the designed system will be applied to and how the database model, for instance, will look like based on the design model.

The Unified Modeling Language (UML) is the core element to represent those models. According to the OMG, UML supports the formalization of abstract, though precise, models of the state of an object, with functions and parameters provided through a predefined interface [OMG01]. In fact, UML provides a wide set of modeling concepts and notations to meet current software modeling techniques. The various aspects of software systems can be represented through UML, what makes it suitable for architecture modeling. As a consequence, UML has been widely used as a basis for software analysis and development.

To model complex domains in UML, new semantics can be added through extension mechanisms that specify how specific model elements are customized and extended within those semantics. In particular, UML profiles make extensive use of UML stereotypes and their attributes (in UML 1.x these are called tag values). Additionally, profiles list a number of constraints that stem from the consistency constraints defined in the UML semantics guide. The language to express these constraints is the Object Constraint Language (OCL) [Gro03, WK03] and is used to specify well-formedness (correct syntax) rules of the metaclasses comprising the UML metamodel. OCL can also be used to specify application-specific constraints in the UML models.

Following this principle, our approach to MDA in this thesis is then to use the UML lightweight extension mechanisms, i.e. profiles. Using these extension mechanisms of the UML, makes our approach consistent with the official MDA white paper [OMG01], which defines basic mechanisms to consistently structure the models and formally express the semantics of the model in a standard way. Moreover, UML profiles define standard UML extensions to describe platform-based artifacts in a design and implemented model. For example, the EJB Profile supports the capture of EJB architecture semantics through the EJB Design Model and the EJB Implementation Model.

In MDA, a mapping is a set of rules and techniques used to modify one model in order to get another model. So as to design and formalize software reliability attributes and analysis properties, we first map them into a profile, according to the mapping mechanism in Figure 2.2.

As a result, following that approach makes possible the expression of semantics and notations of software reliability in a standard way.

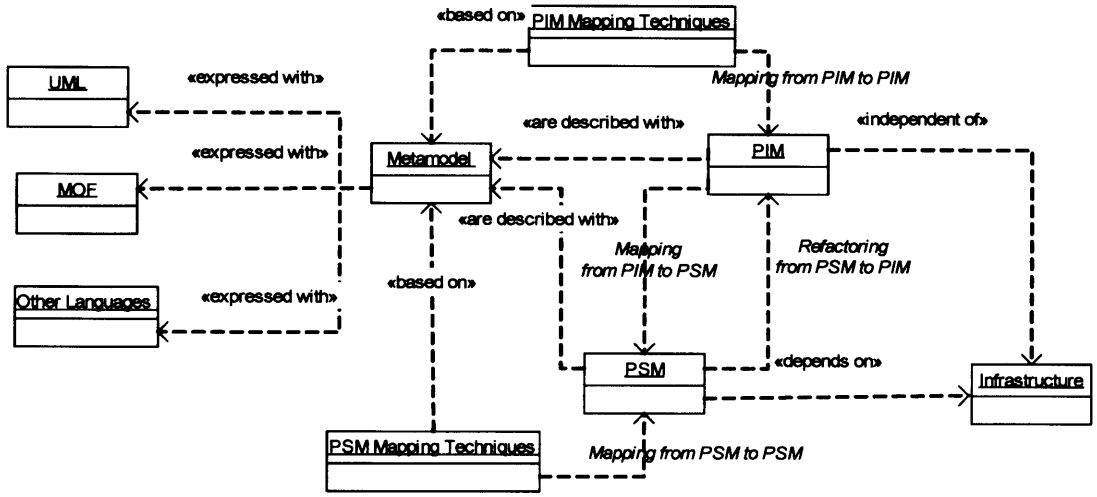


Figure 2.2: MDA Metamodel Description [OMG01]

2.2 The Software Reliability Engineering Process

A software system can be analysed for its quality from many perspectives such as its performance, scalability, maintainability, usability and reliability. In general, software reliability is among the major factors in software quality "... since it quantifies software failures, which can make a powerful system inoperative" [Lyu07]. The study of reliability as the quantification of the operational behaviour of software systems with respect to user requirements is defined as software reliability engineering (SRE) [Lyu07]. The classic SRE process follows the elicitation of four major steps [Lyu96]: (1) *reliability objective*, (2) *operational profile*, (3) *reliability modeling* and (4) *reliability validation.*

A reliability objective relates to what the reliability goal of the software is from the viewpoint of the customer. This reliability objective is related to what kind of system failure the user wants to measure. For this purpose, a well defined view of failure classification must be made. Later in this section, we clarify the class of failures where one could measure the reliability of a system. For each class of failure identified, the reliability requirement can then be defined using an appropriate reliability metric. For instance, a system that can recover without operator intervention could have its reliability measured by the intensity that a failure causes inconvenience to the user [Som01].

The operational profile relates to the information obtained through the system operation on a certain environment. The construction of an operational profile is important in order to select test cases according to the usage of the system [Lyu96]. This is related to concept that the software reliability is affected by software failures rather than the system faults [Som01]. A fault must be executed to cause a failure, otherwise it is just a *dormant* fault [ALR01]. Also, the importance of the operational profile is endorsed by the fact that software reliability is tightly related to the environment where the software is being executed. In particular, a software component rely on various software and hardware resources to be deployed. Software resources comprise those software elements required to execute a component, such as operating systems, middleware, databases and so on. If during the execution of a software component a resource fails, the component requiring that resource will automatically fail, unless fault tolerance techniques are applied.

Reliability modeling is essential to the reliability prediction and estimation process. Most of the reliability modeling approaches attempt to predict software reliability in the later stages of the life cycle [Lyu96], though. The most successful techniques in the literature are probably those classified as Software Reliability Growth Models (SRGM). Those models have been widely used to predict reliability in the later phase of software by modeling the number of faults and the failure rate as testing progresses. As result of those testings, it is expected a growth in the reliability.

Some SRGMs can also be used to estimate software reliability by adding other important factors that affect final software quality. The software reliability estimation determines if a product meets its reliability objective and is ready for release [Lyu96]. To carry this out, failure data should be collected during system testing which are then fit into a reliability model. Although a wide number of reliability models can be found, it is sufficient to consider a dozen models, which provide various estimates of software reliability [Lyu96]. It is important, though, that the number of tests executed is enough in order to have a reasonable confidence over the estimated values [KM97]. However, to achieve the number of tests needed to represent all possible traces the software can traverse is impossible. Alternatively, a usage model with a population set that properly characterizes the system can be used [BL75, Tra95]. Using these statistical methods, the best estimates of reliability are obtained during testing [Lyu96]. If the reliability objective is not met, more testing will be applied in an iterative process.

Finally, reliability validation, as the last step of the software reliability engineering process, consists of comparing the projected reliability with the observed reliability. This validation will then provide a confidence level of the reliability estimate as well as feedbacks to the reliability

engineering process and software enhancement.

As part of all four steps, a component-based software reliability measure has to take into account whether the interface of the components are clearly identified. A component-based software reliability technique is a function of the component reliabilities and therefore, a per component failure basis needs to be clearly distinguished. Unclear separation of the components and their functionalities will render imprecise separation of failure per components.

In this section, we provide background information related to the analysis of software reliability. In Section 2.2.3 we present the principle behind some of the most common software reliability models. As the purpose of software analysis is to verify software properties and check if it is in accordance with the specification, one needs to use fault tolerance in order to assure the reliability will be preserved in the presence of active faults. For this reason, we present in Section 2.2.4 the implementation of fault tolerance, particularly in the distributed objects arena, one of the major targets of MDA. It is not our purpose here to present all the work that has been done in the field of reliability modeling and fault tolerance. There has been plenty of work in both areas and many others coming out. We present here the information relevant to the scope of our work. Last but not least, we present on the last section the notion of scenario, its notations and constructs, the synthesis of a software architecture model through scenarios and the implications when the software specification does not match the synthesized architecture model.

2.2.1 Failure Classification

In order to identify the reliability objective of a system, it is important to identify the classes of failure for a software system. As mentioned previously in this chapter, the reliability objective is related the type of system failure the user wants to measure. The clearest definition for failure we find in the literature is the one given by Avižienis et al. in [ALR01]. They define a system failure as an event that occurs when the delivered service deviates from correct service, and therefore, as transition from correct to incorrect service, with regard to the software system specification.

For the reliability analysis of a software, it is software failures, not software faults, that affect the reliability of a system. A fault is the adjudged or hypothesized cause of an error [ALR01]. When a fault produces an error it is *active*, otherwise it is considered *dormant*. Specifying that a software should have no more than a certain number of faults per line is not meaningful when specifying a software reliability, as pointed out in [Som01]. Those faults could be dormant and therefore means nothing in terms of a system's dynamic behaviour. A system consists of components interaction, and for the study of reliability, the system's state

analysis reveals a more realistic quantification.

Failures can be classified into types and their occurrences are system specific. Avižienis et al. [ALR01] classify failure into clear taxonomic levels: domain(value or timing failures), perception by users (consistent or inconsistent failures) and consequences on the environment (from minor to catastrophic failures). Sommerville [Som01] classifies them with a similar perspective, as depicted in Table 2.1.

Failure Class	*Description*
Transient	Occurs only with certain inputs
Permanent	Occurs only with all inputs
Recoverable	System can recover without operator intervention
Unrecoverable	Operation intervention needed to recover from failure
Non-corrupting	Failure does not corrupt system state or data
Corrupting	Failure corrupts system state or data

Table 2.1: Failure Classification [Som01]

It should be mentioned that the failures in classifications, i.e. Avižienis et al. and Sommerville's, are not meant to be mutually exclusive. Failures from domain, user perception and environment consequences can be combined according to the system and the reliability one wants to measure. For example, a failure can be transient, recoverable and corrupting. For the reliability perspective, it is usually more useful to analyse those permanent or consistent failures. They indicate that once a failure occurs, it will always occur until it is corrected. Therefore, if no fault tolerant mechanism is applied on those failure conditions, all the subsequent system execution is likely to fail. On a transient or inconsistent failure, a system can overcome the failure just by retrying the service. In order to hide persistent failures, for instance, algorithms to make the *failover* of a system could be implemented.

2.2.2 Reliability Metrics

Once the class of failure is identified, the reliability objective can be defined according to a suitable reliability metric. Four kinds of interpretations can be performed for the reliability:

1. Probability of Failure on Demand (POFOD) - Appropriate for systems demanding services at unpredictable or relatively long time intervals and where it would incur in serious consequences not delivering those services. For example, emergency services. Reliability for these systems would mean the likelihood the system will fail when a service request is made.

2. Rate of Occurrence of Failures (ROCOF) - Adequate for systems which services are executed under regular demands and where the focus is on the correct delivery of service. Examples of systems falling in such a category include online bookstore transactions, bank teller system, hotel reservations and many other examples that involve transactions and delivery services. Reliability of such systems represent the frequency of occurrence with which unexpected behaviour is likely to occur.

3. Mean Time to Failure (MTTF) - Applicable to systems used for long time, such as operating systems (not when used to run network servers, that needs to be continuously running) and text editors. Reliability of these systems represent the average time between observed system failures.

4. Availability (AVAIL) - Appropriate for systems delivering services continuously. Examples include network servers and railway signalling systems. Reliability of such systems indicate the probability that the system is available for use at a given time.

2.2.3 Reliability Models

We distinguish software reliability models with respect to those which take the software architecture into account and those that do not. We take the software architecture as a parameter to classify reliability models as the architecture of a software system "is about a holistic view of a system: often a system yet to be developed" [Szy02], which fits properly in the context of software reliability prediction. Moreover, intrinsic to software system architecture is the idea of components interactions as defined by Szyperski in [Szy02]:

> An architecture prescribes proper frameworks for all involved mechanisms, limiting the degrees of freedom to curb variations and enable cooperation. An architecture includes all policy decisions required to enable interoperation across otherwise independent uses of the mechanisms. Policy decisions include the roles of components.

As a result, in the core software architecture lies component interactions and their roles. And by component, Szyperski [Szy02] defines as:

- A unit of independent deployment.
- A unit of third-party composition.
- Without persistent state.

In general lines, a software component is a system element written to a specification (making it reusable) with the purpose to offer a service and is composable with other components.

Most recent reliability models are based on the principle of software architecture but the classic software reliability models are not based on the idea of software architecture. For the architecture-based ones we follow the relatively recent work from Goševa-Popstojanova and Trivedi in [GPT01], while for the non-architecture-based models we follow the classic classification of Musa and Okumoto.

The reliability models that do not consider the software architecture fall in the classification scheme of Musa and Okumoto [MO83]. They are mostly known in software reliability engineering as Software Reliability Growth Models (SRGM), and have five different domains:

1. Time domain - wall clock versus execution time
2. Category - total number of failures in finite or infinite time.
3. Type - either Poisson or binomial distribution of the number of software failures experienced either on debugging, testing or early deployment of the software.
4. Class - For finite failure category, the form of the failure intensity function in terms of time.
5. Family - For infinite failure category, the form of the failure intensity function expressed as the expected number of failures.

Reliability models that fall into Musa-Okumoto classification have as input for their model the number of software failures that occur throughout time of software execution. In particular, we focus on the two major models following the Musa-Okumoto classification: the Goel-Okumoto model and Musa's basic execution time model. These two models became notably adopted by the software reliability community among non-architecture based models, and so we present them here as representative models from this category.[1].

As for the architecture-based software reliability models, they consider software components into their reliability analysis models, making them more suitable to perform what-if analysis and sensitivity analysis for the software system. Reliability models falling in this category reflect the more recent direction of software engineering where software is implemented as a modular structure having components as building blocks. Failure behaviour of architecture-based software reliability models and their components can be specified in terms of their relia-

[1]For other SGRMs that follow in the classification of Musa and Okumoto, chapter 3 of [Lyu96] provides quite a comprehensive in-depth explanation of them.

bilities or failures rates, which can be either constant or time-dependent. Models that explicitly consider software architecture in their reliability models can be classified in two ways :

1. State-based models - models falling in this category use control flow graphs to represent system architecture. They assume transfer of control between components that follow a Markov property, meaning the future behaviour (or state interchangeably) of the system is conditionally independent of the past behaviour. Markov chain models in their turn, can be classified into three sub-categories: Discrete-Time Markov Chain (DTMC) - applicable to software where terminating execution can be clearly identified; Continuous-Time Markov Chain (CTMC) - applicable to continuously operating software applications; and Semi-Markov Process (SMP) - when it is difficult to determine what constitutes a software run or when there is a large number of such runs. The state-based models are also classified into two types: composite and hierarchical. The composite method combines the architecture of the software with the failure behavior into a composite model which is then solved to predict reliability of the application. The hierarchical model, on the other hand, solves first the architectural model and then superimposes the failure behavior on the solution of the architectural model in order to predict reliability [GPT01].

2. Path-based models - models falling in this category compute the software reliability considering the possible execution paths of the program either by testing or by some kind of algorithm. While state-based models analytically account for infinite paths that may come as a results of loops in the software execution, path-based models need to restrict to traverse finite paths by considering average execution time of the application or by restricting the number of paths to be observed during testing.

For other architecture-based models, Goševa-Popstojanova and Trivedi [GPT01] provide a comprehensive survey of the various approaches for those models. In that survey, they include a third kind of architecture-based reliability model, one which is called an *additive model.* Additive models, such as [XW95], implicitly consider the software architecture by modelling the component-based software reliability as a Non Homogeneous Poisson Process (NHPP) process, where the cumulative number of failures and failure intensity function are computed as the *sum of corresponding functions* for each component (hence the reason they are called additive). However, the mere sum may not reflect the dynamic behaviour of components and their interactions, which might compromise the accuracy of the reliability analysis.

Among reliability models based on the software architecture, we discuss in Section 2.2.3.2 a DTMC state-based model: the Cheung model. Cheung's approach for reliability modelling

has influenced many recent proposals on component-based reliability modelling and has a straightforward application to state-based stochastic models. We believe the simplicity and the sound basis of Cheung model account for its widespread use.

In sections 2.2.3.1 we present two of the most known models of non-architecture based as they form the basis of their category. In section 2.2.3.2 we present three of the most popular among architecture reliability models, one of each subcategory: a composite state-based, a hierarchical state-based model, and a path-based model.

2.2.3.1 Non-architecture Based Reliability Models

In this section we summarize two of the most influential non-architecture based reliability models and highlight the reliability metrics that are expected to be obtained through them.

The Goel-Okumoto Model

Amrit Goel and Kazu Okumoto's model [A. 79] has formed the basis for the NHPP models. The NHPP model is a Poisson type model that takes the number of faults per unit of time as independent Poisson random variables. As an NHPP model, the Goel-Okumoto model's major assumption is that the cumulative number of faults per time t, $M(t)$, of a system follows a Poisson process with mean value function $\mu(t)$ [2]. The number of faults detected in each of the respective time intervals is independent for any finite collection of times. This means that the number of detected faults are assumed to have independent increments in disjoint time intervals. Also, it assumes that the *cumulative number of faults* by time detected in a software system to be finite and nondecreasing.

The mean value function $\mu(t)$ of the Poisson process for the fault occurrences for any time t to $t + \Delta t$ is proportional to the expected number of undetected faults at time t. The mean value function can then be obtained in the following form:

$$\mu(t) = N(1 - e^{-bt}) \tag{2.1}$$

where b and N are constants, both greater than zero. In order to determine the values of those two constants, one can use the method of maximum likelihood estimation (MLE) as discussed in either [A. 79] or chapter 3 of [Lyu96]. Notice that N is the expected total number of faults to be eventually detected in the $lim_{t\to\infty}$, meaning that the Goel-Okumoto model is a finite failure model. Assuming faults are fixed as soon as they are detected, the software will become a hundred percent reliable when N is then reached. Therefore, using the Goel-Okumoto model, the *reliability growth* observed can be calculated as follows:

[2] If $\mu(t)$ is a linear function of time, i.e., $\mu(t) = \alpha t$, for some constant $\alpha > 0$, then the Poisson process is a homogeneous process. However, if it is nonlinear, the process is a non-homogeneous one.

$$R(t) = \frac{\mu(t)}{N} = 1 - e^{-bt} \tag{2.2}$$

The way the reliability result can be interpreted for NHPP model, such as the Goel-Okumoto model, will depend on the system and on the basis which failures were extracted [Som01]. But in general, this model can be applied to analyse the system reliability for its ROCOF or MTTF.

The Musa-Okumoto Model

Musa's basic execution time model has had the widest distribution among NHPP software reliability models and was developed by Musa at AT&T Bell Labs [Mus79]. In software reliability, Musa was one of the major proponents of using models to determine software reliability. This model was one of the first to adopt actual software execution time on a computer for the modeling process. This model looks more closely to the computational processing unit of the software rather than elapsed wall-clock time. However, it also provides ways to convert execution time results into calendar time. This is done by relating human and computer resources utilization with execution time.

The Musa-Okumoto model assumes the cumulative number of failures by time t, $M(t)$, and, as a Poisson process, has process with mean value function

$$\mu(t) = \beta_0[1 - exp(-\beta_1 t)] \tag{2.3}$$

where β_0, $\beta_1 > 0$. In order to determine the values of those two constants, β_0 and β_1, one can use the method of maximum likelihood estimation (MLE) as discussed in either [Mus79] or chapter 3 of [Lyu96]. The parameter β_0 is the total number of faults that would be detected in the $lim_{t\to\infty}$, meaning that the Musa-Okumoto model is a finite failure model. This model also assumes the hazard rate for a single fault is constant, which classifies the model into the exponential class.

Finally, using the Musa-Okumoto model, the *reliability growth* observed, once detected failures are corrected, can be obtained as follows:

$$R(t) = \frac{\mu(t)}{\beta_0} = 1 - exp(-\beta_1 t) \tag{2.4}$$

Although the Goel-Okumoto model in Section 2.2.3.1 and the Musa-Okumoto model look very much alike, they differ mostly on the formula to compute their constants. The major difference lies on the fact that Musa's model takes into account the additional CPU hours elapsed since the last software failure was detected. This property makes Musa-Okumoto model more appropriate to consider actual CPU time of software failure rather than mere calendar time.

These measures for the reliability growth we presented in this section and the previous one can be defined as *conditional reliability* since the number of failures experienced are approaching the expected number of failures N. This is mostly useful when a system is in its development phase, as the concern is the next time a failure will happen or similarly, when the software is ready to be released. However, when dealing with a system in its operational phase, the concern is the reliability over a given time interval independently of the number of failures experienced, the *interval reliability*, as described in Chapter 2 of [Lyu96]. Interval reliability is the probability the system experiences no failure during a time interval. This is where the measure of Mean Time Between Failures (MTBF) can play an important role. The greater the MTBF, the more reliable a system is. The MTBF is used in the hardware reliability field when repair or replacement is occurring. The MTBF is computed as:

$$MTBF = MTTF + MTTR \tag{2.5}$$

where MTTF is the Mean Time To Failure and MTTR is the Mean Time To Repair a failure.

2.2.3.2 Architecture-Based Reliability Models

In this section we illustrate architecture-based reliability models through three models representing one of each model type: the Cheung model represents the composite-based model, Kubat represents the hierarchical-based model and Yacoub et al. represent the path-based model.

We also relate them to the reliability metrics that are expected to be obtained through them.

The Cheung User-Oriented Model

As one of the most influential composite state-based reliability models, which is also used in planning and certification of component-based software reliability [PMM93] is the Cheung model [Che80]. Essentially, the Cheung model is a Markov reliability model that uses a program flow graph to represent the structure of the system. Every node N_i in the flow graph of the Cheung model represents a program module and a direct branch (N_i, N_j) represents a possible transfer of control from N_i to N_j. A probability P_{ij} that transition (N_i, N_j) will happen is attached to every directed branch. R_i is the reliability of node N_i. The original transition (N_i, N_j) in the flow graph is then modified into $R_i Pij$, which represents the probability that the execution of module N_i produces the correct result and control is transferred to module N_j. The reliability of the program is, therefore, the probability of reaching the correct termination of the program flow graph from its initial state in the following way: Let $N = \{C, F, N_1, N_2, ..., N_n\}$ be the states of the model, where N_1 is the start state of the program control flow graph, the N_i

are intermediate states, N_n is the last (non-absorbing) state reached in any successful execution of the system, and C and F are absorbing states representing the terminal states Correct (to which there is a transition from N_n) and Fault. Let the transition matrix be M' where M'_{ij} represents the probability of transition from state i to state j:

$$\mathbf{M'} = \begin{array}{c} \\ C \\ F \\ N_1 \\ N_2 \\ \vdots \\ N_n \end{array} \begin{array}{c} \begin{array}{cccccc} \mathrm{C} & \mathrm{F} & N_1 & N_2 & \dots & N_n \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 1-R_1 & 0 & R_1P_{12} & \dots & R_1P_{1n} \\ 0 & 1-R_2 & 0 & R_2P_{22} & \dots & R_2P_{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ R_n & 1-R_n & 0 & 0 & \dots & 0 \end{pmatrix} \end{array}$$

Let M be the matrix obtained from M' by deleting the rows and columns corresponding to the absorbing states C and F. Let S be a matrix such that:

$$S = I + M + M^2 + M^3 + \ldots\ldots = \sum_{k=0}^{\infty} M^k = (I - M)^{-1}$$

where I is the identity matrix with the same dimension as M. Cheung shows that the system reliability is *Rel*= $S_{1,n} \times R_n$, which is the probability of successfully transitioning from N_1 to N_n in any execution, times the probability of successfully reaching C from N_n. Equivalently, Cheung shows that $S_{1,n}$ can be computed as

$$S_{1,n} = (-1)^{n+1} \frac{|M|}{|I - M|} \tag{2.6}$$

where $|M|$ and $|I - M|$ represent the determinant of M and $I - M$, respectively. We refer the reader to Cheung [Che80] for further details on the description and derivation of these formulae.

In general, applying the Cheung model will give the frequency of transition of the system to faulty states, or broadly, the reachability from the initial to the final program module. Considering the DTMC nature of Cheung model, where reliability is computed without regard to time, and its focus on being user oriented information derived from a usage profile of the system, a reliability metric that best suits this model is the ROCOF. However, the limitations of applying Cheung is mostly related to time. Applying Cheung to obtain the MTTF (mean time to failure) mostly depends on whether the components (or module) reliabilities are extracted in terms of MTTF. A transition to the error state in Cheung model in this case would mean the average time the component fails out of the total time unit the system is running.

Some of the limitations of Cheung model have been overcome by [WWC99], where the Cheung model is adapted to obtain the reliability from various software architectures such as pipeline, call-and-return, parallel, and fault tolerance.

The Kubat Model

This model considers the case of a terminating software application, and thus follows a DTMC model, composed of n modules, designed for k different tasks. The relationship between task and modules is that each task may require several modules and the same module can be used for different tasks.

The architecture of the model follows a DTMC model, where task k will call a certain module i with probability $q_i(k)$ and call module j with probability $p_{ij}(k)$ after executing in module i.

Mathematically, the expected number of visits in module i by task k, denoted by $V_i(k)$, can be obtained by solving:

$$V_i(k) = q_i(k) + \sum_{j=1}^{n} V_j(k) p_{ji}(k).$$

And therefore, the reliability of the system as the probability there will be no failure when running task k can be approximated by:

$$R(k) \approx \prod_{i=1}^{n} [R_i(k)]^{V_i(k)} \tag{2.7}$$

The Kubat model has the same potential reliability metrics of the Cheung model, except that it can be extended to accommodate the MTTF of the tasks straightforwardly. In the Kubat model the system failure rate is calculated by the following:

$$\lambda_s = \sum_{k=1}^{K} r_k [1 - R(k)] \tag{2.8}$$

where r_k is the arrival rate of task k.

The Yacoub et al. Model

As a path-based model, this model computes the system reliability through the computation of the possible execution paths of the program on a probabilistic model called Component Dependency Graph (CDG). A CDG is constructed based on scenarios of component interactions and related to run types used in operational profiles. Each node on a CDG is a component and the edges of the CDG represent the transition from one node (component) to another.

The computation of the reliability model considers the component reliability R_i associated to the node n_i representing component C_i with its average execution time t_i. And the transition probability p_{ij} between components C_i and C_j is associated with each directed edge of the CDG. Two additional nodes are included on the CDG: the start and the termination nodes.

The traversal of the CDG can be done in two different ways: (1) when the depth expansion of a path reaches a terminating node, as a natural end of an application execution, or (2) when the summation of execution time of the paths traversed sums to the average execution time of a scenario.

The reliability metrics that can be obtained from the Yacoub et al. model are the same that can be obtained through Cheung one, except that the termination of the system in Yacoub may also happen because the execution time of the system may have been reached.

However, as we discuss further on Section 4.4 in Chapter 4 and on Section 6.7.3 in Chapter 6, we have discovered a limitation of Yacoub et al., in which it is possible to construct CDGs containing imprecise and deadlock states, which are not consistent with the scenario specification and therefore prevents the computation of a reliability metric.

2.2.4 Fault Tolerance

The last part concerning the software reliability engineering process consists of the reliability validation. The validation of the system is dependent on the nature of the system, but in general it consists in monitoring or observing the faults and their consequent failures. Statically, the detected errors can then be eliminated through testing techniques.

At runtime, the system can keep meeting its reliability objective through fault tolerance, which consists in making the software deliver correct service in the presence of faults [ALR01] and it can be implemented in the hardware and in the software level. In general, fault tolerance can be implemented by making redundant hardware or software. The use of techniques for fault tolerance is to ensure that *faults* do not result in *errors* which can result in system *failure* [Som01]. In other words, to avoid an incorrect system state generating an erroneous system behaviour which will prevent a server object from delivering a service expected by its users.

From the hardware perspective, the most used technique is replication of distributed hardware components, which served the basis for the replication of software components. There are two types of replication: *active* and *passive*. In passive replication, there is a primary member and the backup ones which are loaded when the primary member is no longer responsive. Active replication requires voting of the replica. Requests from the members of the replica group are voted, and are delivered only if the majority of the requests are identical.

From the software perspective, there are two different groups [Lyu07]: single version and multi-version. The former relates to error detection, exception handling and transaction management among others. The latter is also known as design diversity and requires that multiple versions of the software are deployed independently. The approach consists in the production of two or more systems aimed at delivering the same service through separate designs

and realizations [AL86]. There are two techniques to implement design diversity: *recovery-block* and *N-version* programming (also coined *multiversion* programming in [KMY91]). The recovery block approach was started by researches led by Brian Randell and Tom Anderson [AK76], [Ran75] since 1975, where alternate software versions are organized in a similar way as dynamic redundancy in hardware. The purpose of the approach is to detect faults at runtime and implement recovery by rollback, where the previous correct state is restored, and by execution of an alternate version. The N-version or mutiversion programming technique was started by researches led by Avižienis since 1976 (see chapter 2 in [Lyu95]). This technique follows the principle of development of multiple independent versions of the software and majority voting among the versions to achieve the tolerance of faults in software. It basically consists in a fault-tolerant software unit that depends on a generic decision algorithm to determine a consensus result from the results produced by other member versions of that unit.

The focus on fault tolerance for this thesis includes mechanisms offered by distributed object oriented platforms to make their server objects achieve a higher degree of service delivery. In Section 2.2.4.1, we delve into those mechanisms present in notable specifications of object oriented middleware for distributed systems. Beforehand, we do not incorporate design diversity into our framework for reliability in model driven engineering. A considerable deal of research would have to be carried out in order to verify if and how design diversity could be applied into model driven engineering. As a result, this could divert our work from its target, where the purpose is to make use of and not conceiving platforms with mechanisms for making services more reliable.

2.2.4.1 Fault Tolerance in Distributed Object Architectures

Current component-based development architectures (CBDA), such as the J2EE and CORBA, address a considerable range of features to maximize the correct delivery of services offered by component-based software systems. Among those range of features, there are the following mechanisms [Emm00a]:

- Replication - Most object-oriented middleware like CORBA and Java-based ones (e.g. RMI and EJB) implement the *at-most-once* reliability request [Emm00a]. By means of replication, a middleware can make their server objects more available by implementing *exactly-once* reliability request. However, the strict implementation of the exactly-once request can be very expensive and difficult to achieve

- Transactions - Usually implements the *atomic* reliability request as transactions avoid a failure occurring in the middle of a requested operation involving at least two distinctive

parties

- Asynchronous message-driven communication - Guarantee that a request has been executed, where request may be executed more than once in case messages acknowledging the execution of the message are lost. Therefore, message-driven communication, e.g. through Java Message Service (JMS), provides support for *k-reliability*, *at-least-reliability* or *best-effort*. Usually, it is possible to choose how reliable the server object will process their request.

- Persistent Server Objects - Server object can be made persistent by keeping track of their state by writing it on a file or by using a database. Objects that have their state persisted are called stateful objects. Implementing stateful server objects is also another way to overcome the at-most-once reliability request of most distributed object middleware

The way those mechanisms are implemented and used at the distributed objects middleware, will depend on the type the middleware fits into. There are two kinds of middleware according to [RSB05]: *explicit* and *implicit* (or *declarative*) ones. Explicit middleware models follow the traditional distributed object programming model like CORBA. The term explicit stems from the need to write to an API to use that middleware service, as shown in Figure 2.3. In this case, the business logic is woven with the middleware API for a particular service.

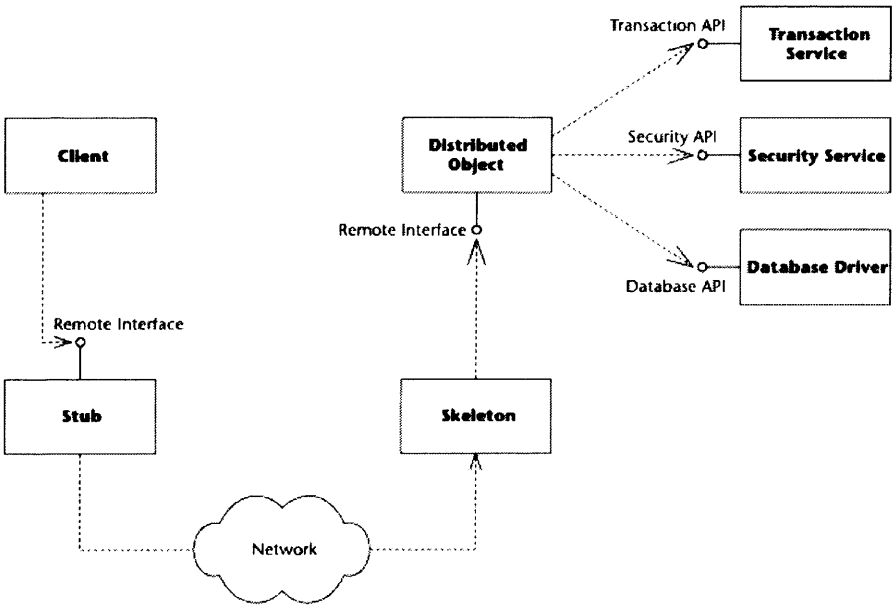


Figure 2.3: Explicit Middleware Architecture [RSB05]

Explicit middleware were the first type of distributed object models to be created and due to complexity in programming for and maintaining those models, the *implicit* middleware

were devised. Examples of implicit middleware include the EJB, the CORBA Component Model (CCM) and the Microsoft .NET. As shown in Figure 2.4, the business logic is completely separate from the middleware services, which are declared on a separate descriptor file. The request interceptor performs the middleware services needed, e.g. persistence and transaction, and then forwards the call to the server object.

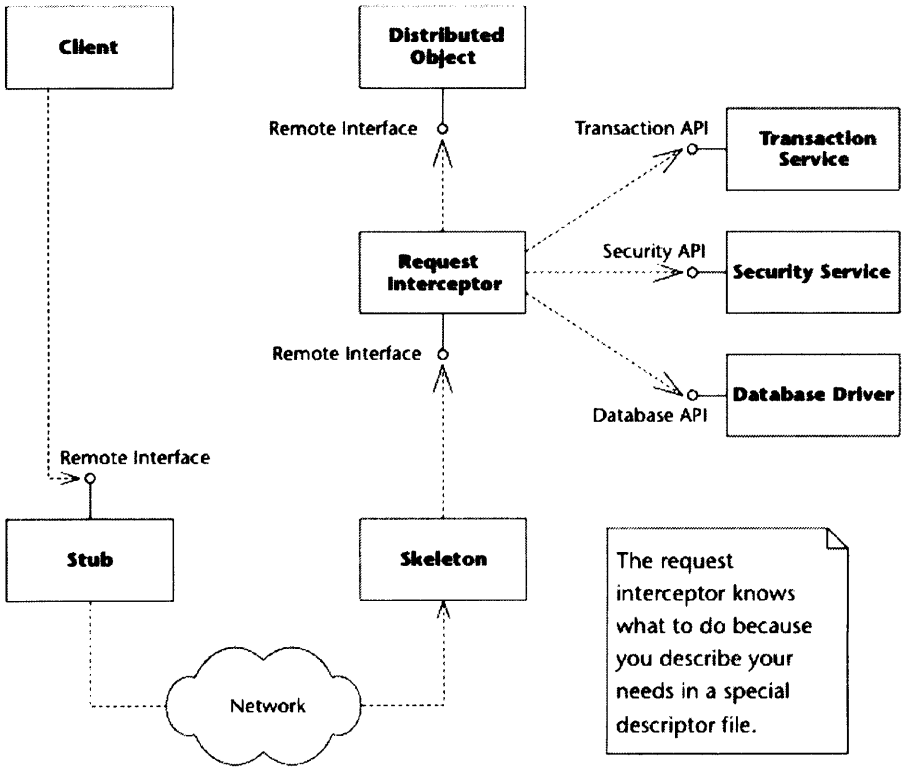


Figure 2.4: Implicit Middleware Architecture [RSB05]

The OMG specifies for their explicit middleware, a set of CORBA-services for the mechanisms to preserve system's reliability in the presence of faults:

- Fault Tolerance Service itself
- Transaction Service
- Event Service
- Persistent State Service

Sun^{TM} provides these mechanisms for their implicit J2EE platform as follows:

- Clustering and the fail-over mechanism
- The two-phase commit protocol and the Java Transaction API (JTA)

- Java Message Service (JMS) and the EJB Message Oriented Bean
- EJB Stateful Session Beans and Entity Beans

In the following two sections, we present an overview of how to make server objects become fault tolerant through CORBA and EJB.

2.2.4.2 Fault Tolerance in CORBA

The CORBA specification for fault-tolerance incorporates quite a comprehensive set of elements in order to "provide a robust support for applications that require a higher level of reliability" [OMG00], compared to those applications where a single backup server suffices. That set of elements specifies fundamental elements a fault tolerant architecture should provide: entity redundancy for each CORBA object; fault detection, notification and analysis for the object replicas; logging; and recovery from faults. The realization of the fault-tolerant CORBA (FT

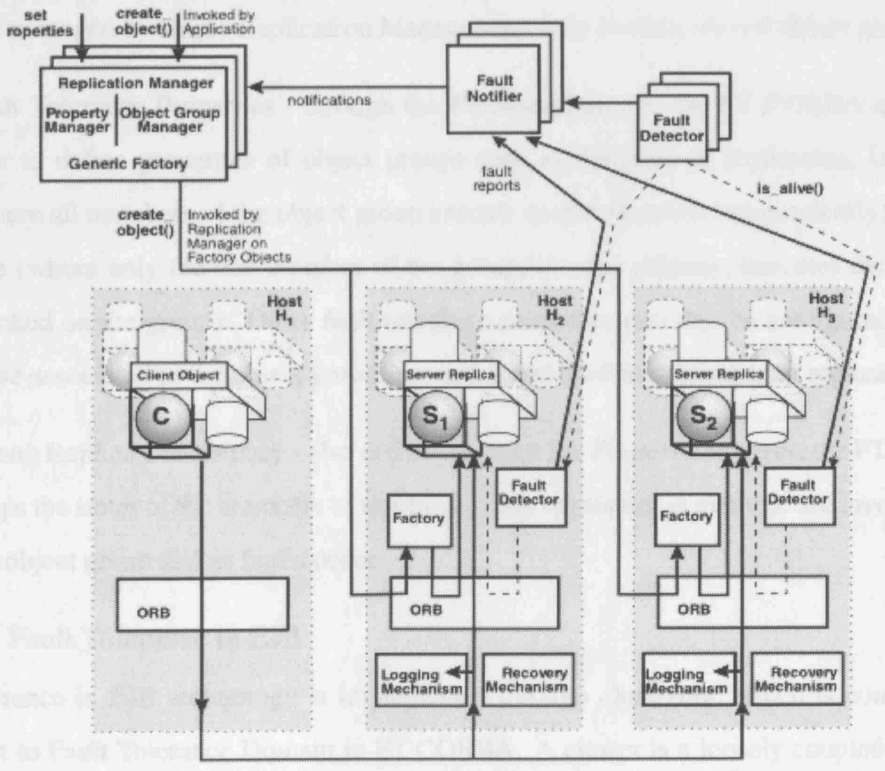


Figure 2.5: A Fault Tolerant CORBA Architecture [OMG00]

CORBA) architecture in Figure 2.5 relies on four basic concepts:

1. Replication and Object Groups - through the *ReplicationManager*, *ObjectGroupManager* and the *GenericFactory* the FT CORBA creates and manages the replicas of a certain server object. On top of the object reference, the replica has an additional reference for the group reference where the replica is part of. The client invokes methods from the

server object, and the members of the server object group does the conventional method execution and reply to the client. This is seen by the client as one object, instead of a set of replicated objects: replication transparency. In case a server replica fails and recovery, the client can not perceive that either: failure transparency. For this failure transparency to take place, four distinct objects are needed: fault detector, fault notifier, logging and recovery. The fault detector detects host or network faults, reports those faults to the fault notifier, which passes the notifications to the *ReplicationManager* and other objects that registered for those notifications. The logging and recovery mechanism will work depending on the replication style (active or passive) chosen through the *PropertyManager* that we present in the FT properties below.

2. Fault Tolerance Domains - to configure security policies and mechanisms to assure the fault tolerance domain has a single security domain. A fault tolerance domain is created and managed by single Replication Manager and may contain several object groups.

3. Fault Tolerance Properties - through the *PropertyManager*, the FT CORBA allows the user to define properties of object groups such as the style of replication, i.e. active (where all members of the object group execute each invocation independently) and passive (where only the one member of the group, i.e. the primary, executes the methods invoked on the group). Other fault tolerance properties can also be configured, such as those associated to the object group, e.g initial and minimum number of replicas.

4. Strong Replica Consistency - also defined through the *PropertyManager*, the FT CORBA keeps the states of the members of an object group consistent as methods are invoked from the object group and as faults occur.

2.2.4.3 Fault Tolerance in EJB

Fault tolerance in EJB technology is implemented through *clustering*, which is conceptually equivalent to Fault Tolerance Domain in FT CORBA. A cluster is a loosely coupled group of server objects that provide unified services to their clients. The clustering process is transparent for the client, and therefore, clients usually have no control over deciding which server processes the clients' request in the cluster. On a 3-tier architecture, where the web container is collocated with an EJB container, a cluster can improve reliability by making feasible scaling up the increase of client demands for a certain service. On the other hand, reliability of a service can also be reduced when extra complexity is added through clustering, making system behaviour less consistent.

In EJB, there are many places where the clustering mechanism can be implemented [RSB05]:

- JNDI - In the Java Naming and Directory Interface context, an EJB vendor could perform clustering by having several equivalent home objects for a given name and spreading traffic across various machines. Some vendors like the JBoss, let one deploy an application to all machines in the cluster at the same time. This mechanism occurs transparently for the client and does not require extra coding in the EJB bean classes.

- Container - The clustering logic can also be implemented on the EJB container. The containers can communicate with each other, transparently to the client, using an interserver communication protocol. This protocol can also be used for load-balancing when, for instance, a Session Bean has filled up its cache. Similarly to the JNDI clustering, this mechanism occurs transparently for the client and does not require extra coding in the EJB bean classes.

- Home Stub - Remote clients have access to this stub object first, which runs at the client's virtual machine. Vendors can implement a kind of smart stub where the stub knows about multiple equivalent copies of the home object. And then, vendors can orchestrate load-balancing and fail-over directly in that smart stub. This is also supported by JBoss clustering and is transparent to the client code.

- Remote Stub - This object represents a specific enterprise bean instance of a client proxy. Clustering through remote stub can follow an analogous approach of Home Stubs, but can be more complex to implement in order to keep the consistency between the instances without disrupting the system.

However, J2EE does not specify an architecture for fault tolerance, but it was implemented by some of the J2EE vendors, where the JBoss is among the most well known. In Chapter 5, Section 5.4.4 we show an example we conducted in JBoss to extract the fault tolerant J2EE architecture elements required in a UML deployment profile for replication in EJB.

In the next section we present an overview of the approach to requirement analysis from scenario specification used as the basis to our reliability analysis modelling.

2.3 From Scenario Specifications to Behaviour Models

Scenario specifications were leveraged by Uchitel et al. in [UKM04] to elaborate system behaviour models. Their contribution has shown to be valuable for making viable quantity anal-

ysis of software systems reliability which consists one of our contributions we delve into in subsequent chapters.

Quantifying software reliability, such as performance and dependability analysis through stochastic behaviour models (or labelled transition systems) is already a common practice in the software analysis community. However, those models are usually too fine grained to represent an accurate view of the software system by its stakeholders. Scenarios, on the other hand, are capable not only to describe the system traces as the behaviour models do but also to depict very clearly the system components designed to provide the intended system behaviour as well as to outline a high level architecture view of the system being described.

Therefore, it is only natural to devise an approach for early reliability prediction for software systems based on such sound work, provided it can be extended to accommodate reliability analysis. Those are the major arguments supporting the extension of the approach from Uchitel et al. to reliability analysis, where stochastic behaviour models are synthesized from annotated scenario specifications, which we develop further on Chapter 4.

For this section, we focus on providing the background regarding the approach of Uchitel et al. in [UKM04]. On the whole, their contribution is twofold: firstly they provide a technique to synthesize scenario specifications into fine grained behaviour models describing the interactions between the components from a global perspective, as presented in Figure 2.6; secondly they implement a technique to iteratively identify implied scenarios, i.e. mismatches between the behavioural and architectural aspects of the specifications.

Before outlining their contributions, we provide their definition for the elements that constitute the basis of their work, primarily including scenario specifications, labelled transition systems, and architecture models extensively referred throughout this thesis.

2.3.1 Scenario Specification

Scenarios are partial descriptions of how components interact to provide system level functionality. Scenario notations such as Message Sequence Charts [ITU96] are used at early stages of development to document, elicit and describe system behaviour. A *scenario specification* is formed by composing multiple BMSCscenarios into HMSCs, possibly from different stakeholders. The underlying notion of scenario composition is that simple scenarios can be used as building blocks to describe new, more complex, scenarios. Simple sequences of behavior are described using BMSCs and composed through the syntactic construction of HMSCs. A BMSC describes finite interactions between components and is formed by vertical lines representing component time lines and horizontal arrows representing interactions between components. We interpret each interaction as a synchronous communication between components. Three funda-

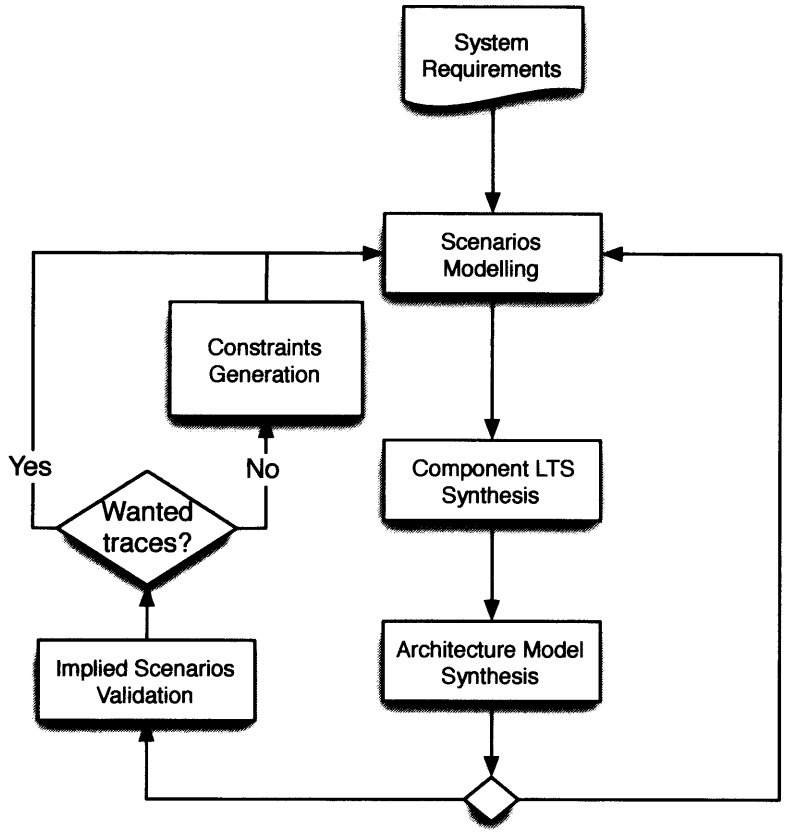


Figure 2.6: Incremental Elaboration of Scenario-Based Specifications and Behaviour Models Using Implied Scenarios from Uchitel et al. [UKM04]

mental constructs for combining BMSCs are *vertical composition* (where two BMSCs can be combined sequentially), *alternative composition* (defining that the system could alternatively choose one of the BMSCs to follow) and *iterative composition* (which composes a BMSC sequentially with itself). A formal definition of a BMSC is presented below:

Definition 2.3.1 (Basic Message Sequence Chart) *– A basic message sequence chart (BMSC) is a structure* b = (E, L, I, M, instance, label, order) *where:*

- E *is a countable set of events that can be partitioned into a set of send and receive events denoted by* send(E) *and* receive(E)*, respectively.*
- L *is a set of message labels*
- I *is a set of instance names*
- M*: send(E)* $\rightarrow$ *receive(E) The pair of send(E) and receive(E), referred as messages in M.*
- instance*:* $E \rightarrow I$ *maps every event to the instance on which the event occurs. Given* $i \in I$*, the set* $\{e \in E \mid instance(e)=i\}$ *is denoted by i(E).*

- label *maps events to labels, requiring that for all (e, e') ∈ M where label(e) = label(e') and if (v, v') ∈ M and label(e) = label(v) then instance(e) = instance(v) and instance(e') = instance(v').*

- order *is a set of total orders* $\leq_i$ *of an instance i, where* $\leq_i$ *corresponds to the top-down ordering of events on* i

The total ordering of events $\leq_i$ presented above model the relation where an event is considered to occur only after all the preceding events on the same instance, following a top-down representation of time. In Uchitel et al. approach, messages in BMSCs are considered to be synchronous, "where the temporal relation between events on different instances is given by the messages of the BMSC" [UKM04].

An HMSC is a directed graph, whose nodes refer to BMSCs and whose edges indicate the acceptable ordering of the BMSCs. HMSCs allow stakeholders to reuse scenarios within a specification and to introduce sequences, loops and alternatives of BMSCs. The semantics of an HMSC is the set of sequences of interactions that follow some maximal path through the HMSC. In Uchitel et al. approach, HMSCs are not presented as having hierarchical feature where an HMSC could have another HMSC as a node. However, according to them, it could be introduced to their approach.

Definition 2.3.2 (High-level Message Sequence Charts) *A high-level message sequence chart (HMSC) is a graph of the form* N, E, s_0 *where* N *is a set of nodes,* E *is a set of edges connecting nodes, and* $s_0 \in N$ *is the initial node. A node n is adjacent to n' if (n, n') ∈ E.*

2.3.2 Labelled Transition Systems

Following the approach of Uchitel et al. once scenarios are modelled, they are translated into label transition systems(LTS) [Kel76] for each component and for the whole system itself, describing their behaviour. Basically, a LTS is a state transition system where transitions have labels. A formal definition of LTS follows below:

Definition 2.3.3 (Labelled Transition System) – *A labelled transition system (LTS), is a structure* P = (S, L,Δ, q), *where:*

- S *is a finite set of states.*

- L = $\alpha(P) \cup \tau$ *where* $\alpha(P)$ *is a set of labels representing the communicating alphabet of P.*

- $\Delta \subseteq (S\backslash\{\pi, \varepsilon\} \times L \times S)$ *defines the labelled transitions between states, where no transition is originated from the states error ,π, or end, ε.*

- q $\in S$ *is the initial states.*

The definition of LTS includes some special elements that are also important in the context of a LTS. There is a special label τ and other two special states ε and π. The τ label models the hidden (from the perspective of other LTSs) internal actions. Those labels are important in the composition of several LTS, where same labels are forced to synchronize. The special states π and ε correspond to *error* and *end* state respectively. We also call them absorbing states, following the terminology of Cheung, where $\Delta = (S\backslash\{\pi, \varepsilon\} \times L \times S)$, i.e. no transition originates from these states.

According to the definition of LTS by Uchitel et al. a deterministic LTS is one where there are no transitions labelled with τ and $(s, l, s_1) \in \Delta$ and $(s, l, s_2) \in \Delta$ implies $s_1 = s_2$. In other words, has no all outgoing transitions of each state in a deterministic LTS have different labels, otherwise it is a non-deterministic LTS. Also, notice that on a deterministic LTS is also minimal, where there are no hidden internal τ labels.

2.3.3 A Simple FSP

For practical reasons, defining an LTS with many states is impractical. In order to turn around the problem, Magee et al. [MKG97][MKG98] created a simple process algebra called Finite State Process(FSP). Therefore, for every LTS there is an equivalent FSP describing the same traces of the corresponding LTS in a concise way through well defined semantics. Magee and Kramer also implemented their contribution in the Labelled Transition Systems Analyser tool (LTSA) [JK99] and an enhanced version of LTSA with scenario specifications was later on implemented by Uchitel et al. [UCKM03].

Considering `x`, `y` and `z` a range of actions and P and Q processes in FSP, we provide a simple example to illustrate the semantics and basic elements of FSP we use in this thesis. Consider the Listing 2.1 as an example of an FSP code. There, we define two processes `P1` and `P2`. The action prefix `x -> Q0` in `P1` defines action `x` and then behaves as described by the auxiliary process `Q0`. In process `P1` there is also the representation of a choice `( x -> Q0 | y -> Q1)` in `Q0`, where the choice delimiter `|` indicates that, while the process can engage in either action `x` or `y`. But only `Q0` or `Q1` will happen once the choice was made between `x` and `y`. A reserved word `END` describes the successful termination of an FSP process, for instance the auxiliary process `Q1` of `P1` and `P2`. On the other hand, an error termination of an LTS process, which models the failure of a process , is described by the reserved

word ERROR.

Listing 2.1: FSP example

```
P1 = Q0,
Q0 = ( x -> Q0
     | y -> Q1),
Q1 = END.

P2 = Q0,
Q0 = (z -> Q1),
Q1 = END.

||Composition = (P1||P2).
```

2.3.4 Parallel Composition

The operator `||` describes the parallel composition of process `P1` and `P2`, meaning that the result is the joint behaviour of both processes where their action execute asynchronously, but synchronize on their shared observable actions, i.e. LTS never synchronize on hidden τ transitions. However, composing LTSs with error termination, represent by the reserved and another one with successful termination will produce an error LTS. A composed error LTS signifies the system failure considering the failure of the process. This feature from composing those kinds of LTS is used to model check if LTSs verify the safety property. On the other hand, if individual processes have only successful termination, the parallel composition will produce an LTS with successful termination. As a result, successful termination of a parallel composition depends on the successful termination of each LTSs. The composed model for the sample FSP processes in Listing 2.1 is presented below in Listing 2.2 and the equivalent LTS depicted in Figure 2.7.

Listing 2.2: FSP Composition for P1 and P2

```
Composition = Q0,
Q0  = (x -> Q0
      |z -> Q1
      |y -> Q3),
Q1  = (x -> Q1
      |y -> Q2),
Q2  = END,
Q3  = (z -> Q2).
```

2.3.5 Architecture Model

As previously mentioned, the work of Uchitel et al. [UKM04] , they contribute the synthesis of behaviour model that takes into account the architecture and the behaviour in the scenario

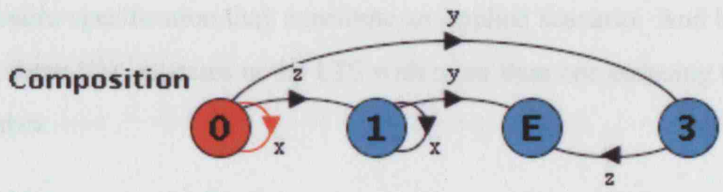


Figure 2.7: The Composed P1 and P2 in LTS

specification. Considering a system is composed by its components and their interactions, the model in the work of Uchitel et. al.follows that principle. The system model in their approach is therefore the parallel composition of the collection of LTSs, where each of those LTS models the component in the scenario specification. Also, the composed model exhibits all those traces described in the scenario specification. To that composed model they attributed the name of *architecture model.*

2.3.6 Model Synthesis Through Scenarios

Now that we have provided the major background definitions, we can delve into the first major contribution of Uchitel et al. which consists on the synthesis of system's behaviour models from scenario specification. On general lines, the approach consists of the following steps:

1. For each component C_i and each BMSC S_j, a *labelled transition system* (LTS) $C_i\_S_j$ is constructed by projecting the local behaviour of C_i within S_j. In particular, each message with an action a that C_i sends or receives in S_j is synthesised as a transition with action a in $C_i\_S_j$, and the sequence of transitions in $C_i\_S_j$ corresponds with the sequence of messages sent or received by C_i in S_j.

2. For each component C_i, the set of Label Transition Systems (LTSs) constructed for C_i in step 1 are composed into a *component LTS* for C_i according to the structure of the HMSC, with hidden transitions (τ actions) linking the final state of $C_i\_S_j$ to the start state of $C_i\_S_{j'}$ whenever there is a transition from S_j to $S_{j'}$ in the HMSC. The resulting LTS includes a new start state corresponding to the start state of the HMSC.

3. Each component LTS constructed in step 2 is reduced to a trace-equivalent minimal, deterministic LTS. By minimal we mean that the LTS traces are the minimal set provided by any model that exhibits all specified traces and also preserves the component structure with its interfaces. This is consistent with the delayed choice semantics of the ITU MSC standard [ITU96]. We will see in the next section that the traces that are exhibited in the composed model of the system (the architecture model) and are not specified explicitly in

the scenario specification they constitute an implied scenario. And by deterministic we simply mean that no states in the LTS with more than one outgoing transitions have the same label.

4. The architecture model for the system is taken as the parallel composition of the minimised component LTSs constructed in step 3.

The second contribution of Uchitel et al. is the technique to iteratively identify implied scenarios, which we succinctly approach in the next section.

2.3.7 Implied Scenarios

Scenarios describe two aspects of a system: (1) a set of traces the system is intended to exhibit, and (2) the components that will provide system level functionality and their interfaces (the messages these components can use to interact between each other to provide system level functionality).

It has been shown [AEY00, UKM04] that given a scenario specification, it may be impossible to build a set of components that communicate exclusively through the interfaces described and that exhibit only the specified traces when running in parallel. The additional unspecified traces that are exhibited by the composed system are all called *implied scenarios* and are the result of specifying the behavior of a system from a global perspective yet expecting it to be provided by independent entities with a local system view. If the interaction mechanisms do not provide components with a rich enough local view of what is happening at a system level, they may not be able to enforce the intended system behavior. Effectively, what may occur is that each component may, from its local perspective, believe that it is behaving correctly, yet from a system perspective the behavior may not be what is intended.

Implied scenarios indicate gaps in a scenario-based specification. They can represent intended system behaviour that was missing from the inherently partial scenario specification or undesired behaviour that should be avoided by changing the architecture of the system. Hence, implied scenarios need to be validated (identifying them as *positive* or *negative* system behaviour) and the scenario specification elaborated accordingly with additional BMSCs added for positive scenarios and constraints added to the synthesized model to capture negative scenarios.

2.4 Summary

In this chapter we provided the theoretical grounding required to grasp the following chapters. First, we presented in Section 2.1 an overview about MDA and the approach we follow to ac-

complish a UML profile for reliability, which is elaborated further on Chapter 5. Second, we presented in Section 2.2 an overview about the software reliability engineering process, including notions on failure classification, reliability metrics, reliability models and fault tolerance in order to keep the system meeting its reliability objective at runtime in spite of active faults. Finally in Section 2.3, we presented a technique by Uchitel et al. [UKM04] to model synthesis of software systems through scenarios, accounting for what is called *implied scenarios*.

In the next chapter we present a guiding example that raises the motivation underlying the contributions of our thesis.

Chapter 3

Brief Overview of Our Approach

The importance of obtaining software quality attributes early in the software engineering cycle has gained wider interest on the research arena due to the increasing demand from industry. Reduction in costs and time to engineer a software are among the most notable reasons. The focus of this thesis is model-based reliability prediction, as a software quality attribute. There are numerous ways to predict or measure software reliability. In the past, however, a great deal of effort in the research of software reliability focused on modeling the reliability growth of the software after debugging. Those models treated the software as a monolithic whole without concern to its internal structure. With the advent of component-based software development, the need to have software quality analysis that can accommodate such structures has become more evident. In this chapter, we introduce an illustrative example with clear component interfaces, from which we want to analyse its software reliability. We raise the major questions from the stakeholder point of view and then delineate our approach to obtain an early estimate of the system reliability, that we tackle throughout our thesis.

3.1 A Guiding Example

It is common practice nowadays to develop software systems as component parts of larger applications, instead of all-inclusive applications. Therefore, analysing the software quality requires models that include software components and how they weave together. On the whole, this is the motivation behind architecture-based approach for quantitative assessment of software systems. As architecture-based models become available, stakeholders are then able to analyse quantitatively the software throughout its life cycle and to invest on those components and interfaces that are more critical to the system. We propose in this thesis an architecture-based software reliability technique to address those issues. In order to do that, we start with a motivating example where we characterize what constitutes the informal specification of a steam boiler control system. This example has been presented at Uchitel et al. [UKM04] and extensively used in previous papers we published as an illustrative example.

The boiler control system includes a steam sensor that produces a control signal in the presence of steam. A *Control* module is associated with the water reservoir and coupled to the *Steam Sensor* and the heating element, the *Actuator*, for activating and deactivating the heating element in response to the control signal produced by the *Sensor*. An important requirement though is that the system works correctly as the quantity of water present when the steam boiler is working has to be neither too low nor too high. If this requirement is not followed, the steam boiler or the turbine in front of it might be seriously affected. We depict the example in Figure 3.1. Based on the description above, there are three immediate components we can identify as part of the boiler control system: Control, Sensor, Actuator. We include another component, generically called *Database* to store the information produced by the Sensor.

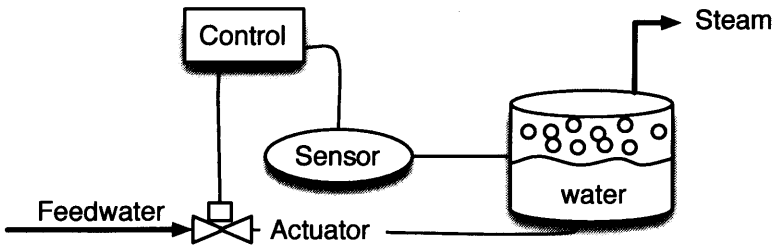


Figure 3.1: The Architecture a Hypothetical Steam Boiler.

Based on those requirements for the boiler system, we build scenarios in order to elicit and analyse requirements of the boiler control system following the background on scenario specifications presented in Chapter 2, i.e. that of BMSCs and HMSCs. A well known representation of scenarios derives from the UML specification where HMSC are usually represented

as Activity Diagrams or as Interaction Overview Diagrams (introduced in UML 2) whereas the BMSC is represented by the Sequence Diagrams.

We present the scenarios for our illustrative example in Figure 3.2. In the top left of the Figure, there is the Boiler HMSC represented as an Activity Diagram where nodes are the four BMSC scenarios: *Initialize*, *Register*, *Analyse*, and *Teminate*.

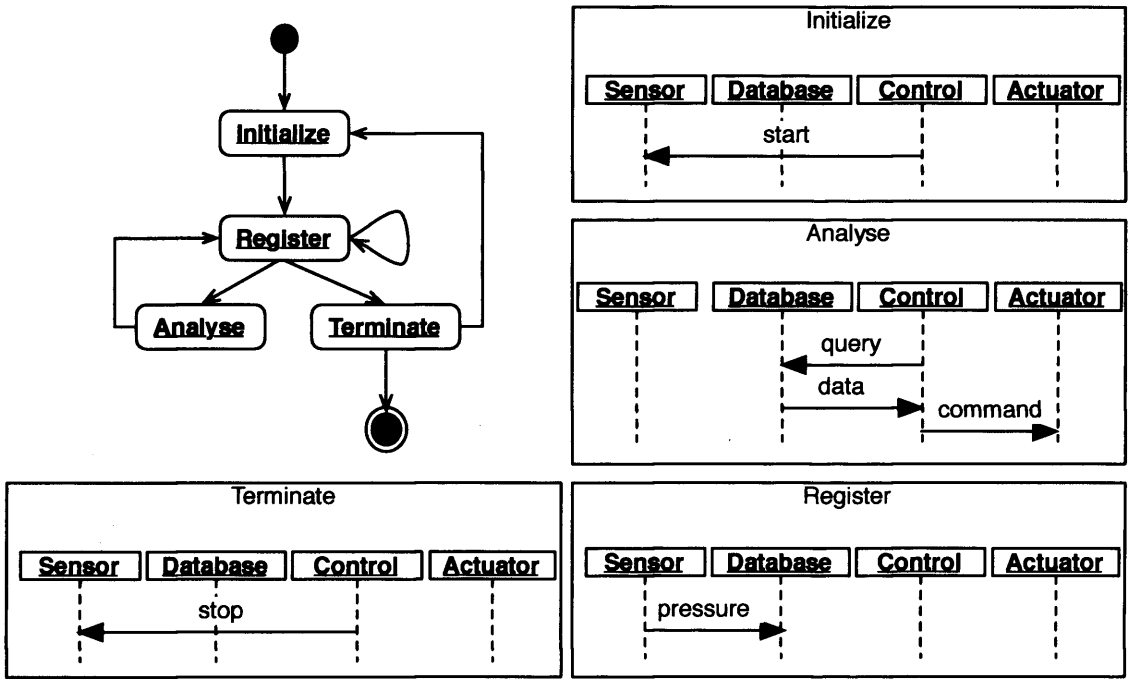


Figure 3.2: The Scenario Specification for the Boiler.

From the reliability perspective, we want to know a preliminary quality value of the steam boiler taking into account the architecture of the software composed by components. Scenario specifications are very suitable to express system requirements and to elicit the software architecture with the interactions between the components. Additionally, one may want to know those elements composing the scenario specifications that impact more on the system reliability so that a fault tolerant model of the system can be already devised. Other questions related to what-if analysis may also come naturally. For example, a valid question to make at this stage is what are the components or scenario transitions that have a more significant impact on the system reliability based on the models?

To be able to answer those question we need reliability models that are applicable in the earlier phases of the software engineer. Those models are (or should be) mostly based on the software architecture and component-based software engineering. However, there are not many of these models available and therefore, experts in the area like Michael Lyu have recently con-

sidered them as an area for possible future direction [Lyu07]. A software architecture consists of software components, and therefore, modeling those architecture for reliability needs to take into account the components relationships and their properties. Most of the reliability modeling approaches attempt to predict reliability in the later stages of the software life cycle [Lyu96], though. The most successful techniques in the literature are probably those classified as Software Reliability Growth Models (SRGM), which require the expertise in reliability analysis, unsual in the software development arena.

3.2 An Approach to Early Software Reliability Estimation

Early estimate of the system reliability can be obtained by enriching the scenario specifications and subsequently transforming them into enhanced behaviour models. In order to do that, we extend that approach of Uchitel et al. [UKM04] we introduced in Chapter 2 by first annotating the scenario specifications with information to explore the system reliability at the modelling level. Then, once the scenario specifications are synthesized into a fine grained model of the system, we translate that synthesized model into the probability that the system delivers the service correctly, i.e. the system reliability, through the probability the system reaches the correct final output of the system. In the end of that process, we are able to abstract away the complexity of reliability modelling through models that reflect the view of the system from the stakeholders perspective and anticipate information that can be used to enhance the system reliabiity.

This is one of the motivation upon which we build the work on this thesis and is depicted in Figure 3.3. In the left hand-side branch of the Figure 3.3, following from *System Requirements*, we present the process of computing reliability from the annotated scenario specifications. Those specifications are then translated into the representation of the component behaviour in LTS, with probability weights annotating the transitions between states. A stochastic architecture model of the system is constructed as a result for the parallel composition of synthesized component behaviour models. That synthesis is done using the labelled transition analyser tool (LTSA), which can model check the system for deadlock, safety and liveness properties [JK99].

In the right hand-side of Figure 3.3, following from *Reliability Domain Modelling*, we depict the other motivation of our work where we contribute the elaboration of a standardization process of reliability modelling following the standard MDA approach. Following that process, it is possible to semantically integrate software reliability modeling from design to analysis and to accompany evolution of software system reliability in accordance with the standard model

driven engineering process.

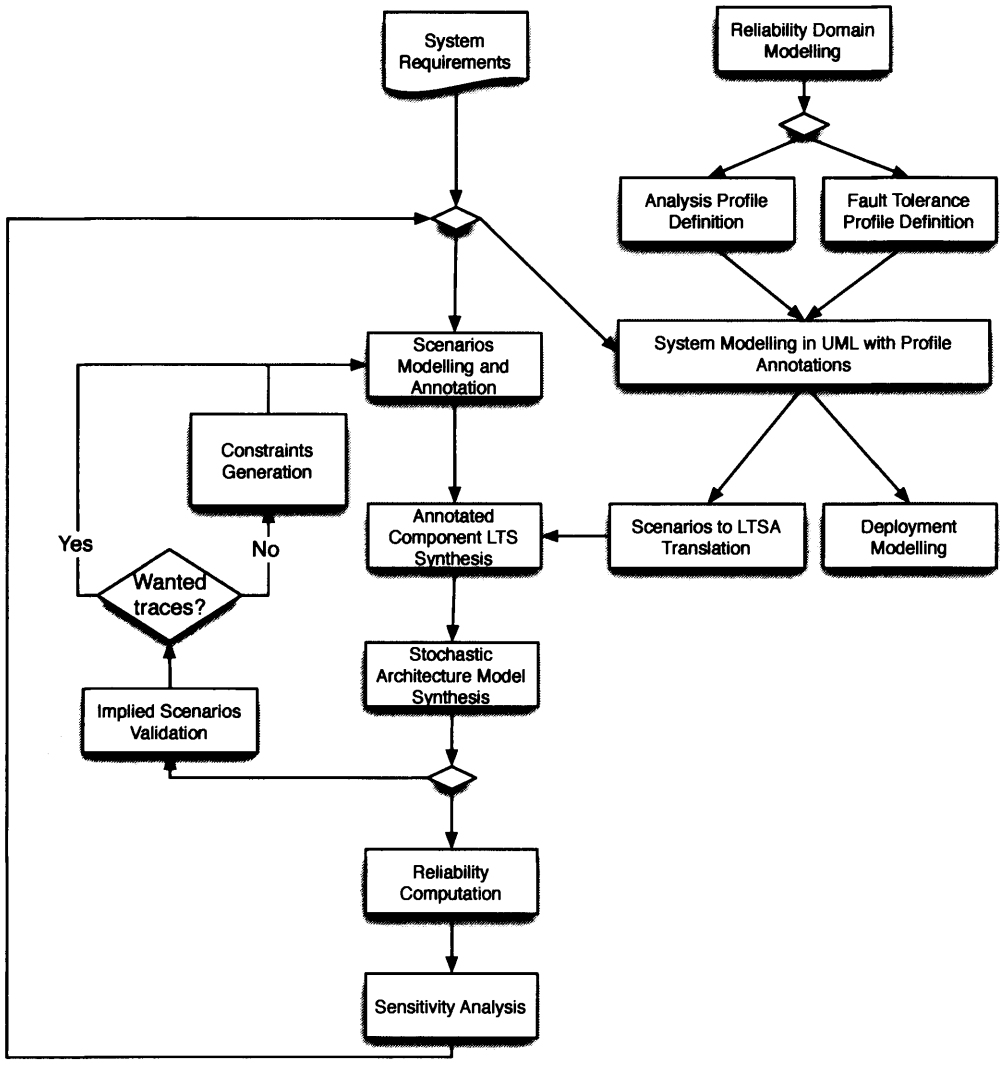


Figure 3.3: The Model Processing Framework for Reliability Prediction.

To carry out our approach, we first delineate the assumptions that constitute the basis on which the extended work can be done. Four key assumptions underlie this extension:

1. The transfer of control between components has the Markov property, meaning that the transition from one execution state to another is dependent only on the source state and its available transitions and not on the past history of state transitions. This is a traditional assumption that simplifies the work on reliability analysis and it greatly simplifies the computation of reliability estimates.

2. Failures are independent across transitions. Again, this assumption simplifies the computation of reliability estimates.

3. A message from component C to component C' represents an invocation by C of a ser-

vice offered by C'. The reliability with which this service is performed is thus the reliability of C', $R_{C'}$. Additionally, the execution time of the invocation is assumed to be so short as not to be a factor in the component's reliability. In other words, $R_{C'}$ is the probability of successful completion of an invocation of any service offered by C', irrespective of the execution time of the service. This assumption is simply a modeling choice that is made without loss of generality. For instance, we could just as easily accommodate method-level reliabilities, and/or communication reliabilities (as is done, for instance, in Yacoub et.al [YCA99])

4. There is only one initial and one final scenario for the system in the HMSC. Multiple initial and final scenarios can be combined by introducing a *super-initial* and a *super-final* scenario, analogously to the *super-initial state* and *super-final state* proposed by Wang et al. [WWC99].

Once those behaviour models are built, some what-if analysis can be performed. As the boiler system is comprised of four components, the system stakeholders may be interested on what impact each component reliability has on the overall system reliability. As mentioned in [Som01], because highly reliable software is expensive, it is usually sensible to assess the reliability of each sub-system separately rather than impose the same reliability requirement on all sub-systems[1]. This saves effort on placing high demands for reliability on sub-systems unnecessarily. Another question that can be potentially answered from those models is: how many system executions are required to be performed in order to obtain the predicted reliability, i.e., what is the asymptotic behaviour of the system reliability?

3.2.1 Implied Scenarios

In the contribution of Uchitel et al. in [UKM04], they also show it may be possible to have mismatch between the synthesized architecture model and the scenarios specifications. That mismatch indicates that the synthesized architecture model contains traces the system could potentially execute and were not perceived while designing the system scenarios. From the point of view of software reliability, knowing whether nothing 'bad' or unexpected happens during software execution is clearly important.

In the example in Figure 3.2, we see that the Boiler Control System is expected to exhibit a trace *"start, pressure, query, data, command ..."* and that component Control interacts with *Database* only through messages *query* and *data*.

The Boiler Control System of Figure 3.2 has implied scenarios, Figure 3.4 shows one of

[1]From component-based software development approach, those sub-systems can be interpreted as components.

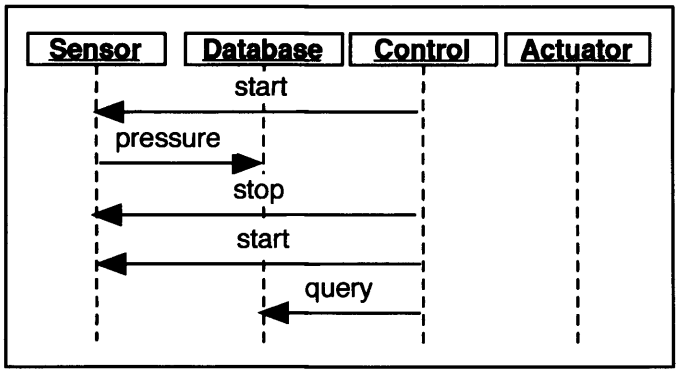


Figure 3.4: Implied Scenario Detected

them. From the specification it is simple to see that after initialising *Sensor* there must be some pressure data registered into the *Database* before any queries can be done. However, in the implied scenario of Figure 3.4 a query is being performed immediately after start.

Why is this occurring? The cause is an inadequate architecture for the traces specified in the MSC specification. The *Control* component cannot observe when the Sensor has registered data in the *Database*, thus if it is to query the *Database* after data has been registered at least once, it must rely on the Database to enable and disable queries when appropriate. However, as the *Database* cannot tell when the *Sensor* has been turned on or off, it cannot distinguish a first registration of data from others. Thus, it cannot enable and disable queries appropriately. Succinctly, components do not have enough local information to prevent the system execution shown in Figure 3.4. Note that each component is behaving correctly from its local point of view, i.e. it is behaving according to some valid sequence of BMSCs. The problem is that each component is following a different sequence of BMSCs! The *Sensor*, *Control* and *Actuator* are going through scenarios *Initialise*, *Register*, *Terminate*, *Initialise*, *Analyse*, *Register*. However, the *Database* is performing *Initialise*, *Register*, *Analyse*, *Register*.

3.3 After Reliability Computation

Later in the software development phase, during the testing of the software, it may also occur that the reliability of the system is not the one required or intended. Also, it may be desirable to find out some modelling elements that impose higher impact on the system reliability.

Through our approach, we carry out two different empirical sensitivity analyses of the system reliability: (1) as a function of the components' reliability and (2) as a function of the transition probability between scenarios. Once the sensitivity analysis is carried for those model elements, it is possible to devise strategies, especially fault tolerance ones, to assurance the software reliability meets its objective.

Once the analysis cycle is concluded, the results can be re-applied into UML with adequate profile annotation. And finally the deployment step can come into play.

3.4 Summary

In this chapter, we have sketched out an overview of our approach to early software reliability analysis following model driven engineering standards. We delineated the requirements elicited for the illustrative Boiler example and briefly presented the framework for our approach. In the following chapters we further elaborate on those steps.

Chapter 4

A Technique for Scenario Based Reliability Prediction

Scenario specifications, which describe component message interactions and high-level architecture view of software systems, are a popular means for capturing behavioural requirements. Because a system consists of a set of interacting components such that the interactions can reveal faults [ALR01], modeling and annotating these interactions appropriately can assist us in predicting software reliability. However, there are not many available sound reliability analysis techniques that exploit systems' dynamic models, such as scenarios. Provided component reliability and scenario transition probabilities are available, scenarios specifications can be used to perform early reliability assessment. In this chapter, we delve further into the approach we outlined in Chapter 3 to predict software reliability based on scenario specifications. We extend the approach of Uchitel et al. [UKM04] first to synthesize probabilistic behaviour models (LTSs) for each component. Then we make those LTSs minimal and deterministic, which enables the reuse of generative parallel composition of LTSs to build a probabilistic behaviour model of the whole system. Finally, we use Cheung's reliability model to compute the system reliability of that probabilistic behaviour model and to conduct further sensitivity analysis of the system.

4.1 Probabilistic Scenario Specifications

Our approach for reliability prediction introduces new attributes into scenario specifications, i.e. the probability of transition between scenarios (as nodes of an HMSC) and component reliability (added to instances of a BMSC). Following this approach, we extend the definition of a BMSC and and HMSC from Uchitel et al. and redefine them into a probabilistic BMSC (pBMSC) and a probabilistic HMSC (pHMSC), respectively.

Definition 4.1.1 (Probabilistic Message Sequence Charts) – *A probabilistic basic message sequence chart (pBMSC) is a structure* b = (E, L, I, M, instance, label, order, R, P) *where:*

- E *is a countable set of events that can be partitioned into a set of send,* send(E), *and receive,* receive(E), *events.*
- L *is a set of message labels*
- I *is a set of instance names.*
- M*: send(E)* $\rightarrow$ *receive(E) The pair of send(E) and receive(E), referred as messages in M.*
- instance*:* $E \rightarrow I$ *maps every event to the instance on which the event occurs. Given* $i \in I$*, the set* $\{e \in E \mid instance(e)=i\}$ *is denoted by i(E).*
- label *maps events to labels, requiring that for all (e, e')* $\in M$ *where label(e) = label(e') and if (v, v')* $\in M$ *and label(e) = label(v) then instance(e) = instance(v) and instance(e') = instance(v').*
- order *is a set of total orders* $\leq_i$ *of an instance i, where* $\leq_i$ *corresponds to the top-down ordering of events on* i
- *R:* $I \rightarrow [0,1]$*, the reliability of an instance* $i \in I$*.*
- *P:* $E \rightarrow [0,1]$*, the probability an event e is successful P(e), where e* $\in$ receive(E)*, is the reliability of instance i, R(i), and instance(e) = i.*

We mentioned previously in Chapter 3 the following assumption for the reliability of a service:

> A message from component C to component C' represents an invocation by C of a service offered by C'. The reliability with which this service is performed is thus the reliability of the invoked component C'.

In the approach of Uchitel et al. each event ($e \in E$) of a BMSC can be distinguished into send event, *send(E)* and receive event, *receive(E)*. Those events are then mapped to instances ($i \in I$) of the components through the function $instance(e) = i$. We exploit this distinction in order to define the terms for reliability of a component i as *R(i)* and probability of a successful event e *P(e)*, satisfying our assumption where the reliability of service is the reliability of the invoked component to perform that service.

Once the definition of a pBMSC is formalized, we can now redefine a probabilistic HMSC. We extend the definition of Uchitel et al. [UKM04] to include the probability of transition between scenarios as follows:

Definition 4.1.2 (Probabilistic High-level Message Sequence Charts) *A probabilistic high-level message sequence chart (pHMSC) is a graph of the form* N, E, s_0, PTS *where* N *is a set of nodes,* E *is a set of edges connecting nodes,* $s_0 \in N$ *is the initial node. A node n is adjacent to n' if (n, n') ∈ E. The probability of transition between scenarios is defined as PTS:* $E \rightarrow [0,1]$, *satisfying:*

$$\forall n \in N, \sum_{e \in S(n)} PTS(e) = 1$$

where S: N $\rightarrow 2^E$, and $S(n) = \{e \in E \mid \exists n' \in N . e = (n, n')\}$ $S(n)$ is the set of edges for the adjacent nodes of a node n. The transition probability PTS_{ij} is the probability that execution control transfers directly from scenario S_i to scenario S_j. This information would be normally derived from an operational profile for the system [Mus93]. Thus, from scenario S_i, the sum of the probabilities PTS_{ij} for all successor scenarios S_j is equal to one.

4.2 The Methodology

The methodology of our technique consists in first annotating scenario specifications with probabilistic properties and then the use of probabilistic labelled transition systems (LTS) synthesised from those scenarios.

Following the framework we outlined in Chapter 3, the reliability prediction consists of six major tasks: (1) annotation of the scenarios, (2) synthesis of the probabilistic components LTS, (3) synthesis of the probabilistic model, (4) system reliability prediction, (5) implied scenario validation and (6) sensitivity analysis.

We delve into further details for those tasks in the next sections as follows: in Section 4.2.1 we develop task one, in Section 4.2.2 we approach tasks two and three, in Section 4.2.3 we carry out task four, in Section 4.2.4 we conduct task five to reliability prediction and finally in Section 4.2.5 we accomplish task six.

4.2.1 Annotation of Scenarios

The first task of our technique consists in annotating the scenario specification with the probability of transitions between scenarios in the pHMSC edges with the PTS and with the reliability of the components R in the pBMSC. Let us take the Boiler example we introduced in Chapter 3, to assist us on presenting our methodology. In order to do this, we first annotate adequately the scenario specification for the Boiler as depicted in Figure 4.1.

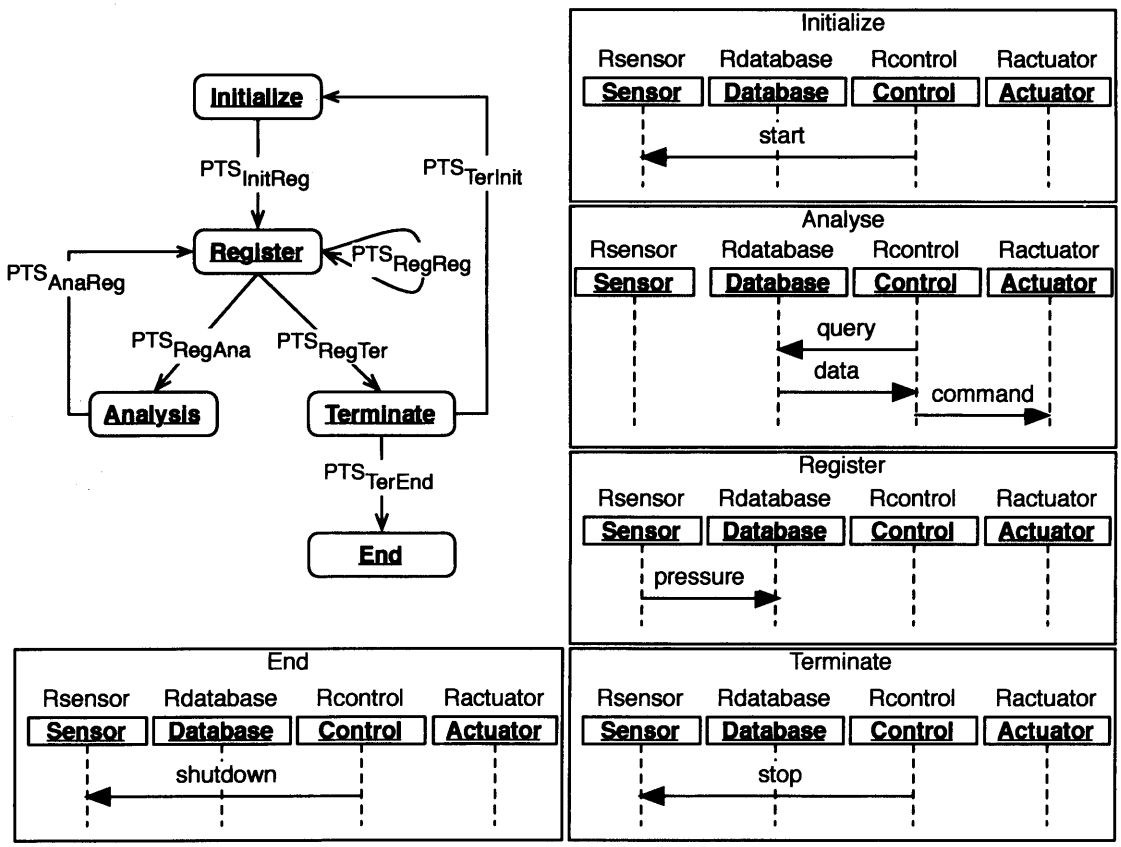


Figure 4.1: Annotated Scenario Specification for the Boiler.

$PTS_{InitReg}$	PTS_{RegReg}	PTS_{RegAna}	PTS_{RegTer}	PTS_{AnaReg}	$PTS_{TerInit}$	PTS_{TerEnd}
1.0	0.7	0.2	0.1	1.0	0.5	0.5

Table 4.1: Scenario Transition Probability Values for the Boiler

Rsensor	*Rdatabase*	*Rcontrol*	*Ractuator*
0.99	0.999	0.95	0.99

Table 4.2: Component Reliability Values for the Boiler

The probability of transition from scenario S_i to scenario S_j of an HMSC we call PTS_{ij}. These probabilities are annotated on the corresponding edges of the HMSC, as shown on the

HMSC of Figure 4.1. For the purposes of illustrating our method on the Boiler example, we use the values depicted in Table 4.1 for the PTS_{ij}. Those values are based on the assumption that the system executes the scenario *Register* (which causes sensor readings to be entered into the database) far more frequently than the scenarios *Analyse* and *Terminate*, and that when it does execute *Terminate* there is an equal probability of reinitialising and shutting down. Notice that we introduced a new scenario into the Boiler system: *End*. We introduced this scenario so that the Boiler system have a final termination, where no other scenario can be executed. This condition is important in order to bring the system to one final state, so that we can compute the probability the given system will successfully reach that final state. We elaborate further on this in Section 4.2.3.

The component reliabilities R_C are annotated on the BMSCs, as also shown in Figure 4.1, and their values presented in Table 4.2 considering the reliability of those components as the probability of success for service invocations made to those components. Without loss of generality, this thesis uses coarse-grained, single values for the overall component reliabilities; in general, we could associate reliabilities with individual messages and/or segments of component timelines.

The values on Table 4.2 for the reliability of the components reflect the assumption that the *Database* is a highly reliable commercial software product, that the *Sensor* and *Actuator* are components whose hardware interface to the sensed/actuated phenomena will eventually fail, and that *Control* is a complex software subsystem that still contains latent faults.

4.2.2 Synthesis of the Probabilistic LTS

The second task of our method is to synthesise a probabilistic labelled transition system (shortly, pLTS), from the annotated scenario specification. We aim at obtaining first a pLTS for each component and then an architecture model synthesized from the parallel composition of those components pLTSs. Before undertaking the synthesis of pLTS from scenario specifications, we present the definition for a pLTS, introduced by Larsen and Skou in [LS92].

Definition 4.2.1 (Probabilistic Labelled Transition System) – *A probabilistic labelled transition system (pLTS), is a structure* Γ = (S, L,Δ, q), *where:*

- S *is a finite set of states.*
- L = $\alpha(\Gamma) \cup \{\tau\}$ *where* $\alpha(\Gamma)$ *is a set of labels representing the communicating alphabet of* Γ.
- $\Delta \subseteq (S \backslash \{\pi, \varepsilon\} \times P \times L \times S)$ *defines the labelled transitions between states with weights* $p \in P$, *the transition probability function.*

- q $\in S$ *is the initial states.*

The major difference between a pLTS and a LTS, presented previously in Chapter 2 is that the former specializes the latter by introducing a probability function to label transitions between states, where P satisfies the following property:

$$\forall s \in S, \sum_{l \in L} \sum_{s' \in S} P(s, l, s') = 1.$$

Therefore, the pLTS constitutes a probabilistic model, where the sum of all the probabilities for all the outgoing transitions for every state of the pLTS sum to one.

The synthesis of pLTS we carry out for each component is an extension of the synthesis approach of Uchitel et al. [UKM03], presented in chapter 2. Our extension of this approach exploits recent probabilistic extensions to the LTS formalism [AFMB03] and involves enhancements to each step listed in Chapter 2, Section 2.3.6. The enhancements have the effect of mapping the probability annotations of the scenario specification into weights for transitions in the synthesised architecture model.

The algorithm to implement steps 1 and 2 is presented in Listing 4.1. A general purpose of the algorithm is to translate scenarios into FSP specification with probabilities derived from the annotated scenario specifications. For each component, an FSP process with weights on each action of that process is first synthesized. Each process exhibits the same behaviour of the corresponding component in the pBMSC with its according weights.

Listing 4.1: The Synthesis Algorithm from Annotated Scenarios to Probabilistic FSP

```
1  void SynthesiseComponent(Specification S, Component c) {
2    print ``\\-----'' + c.name()+ ``-----'';

4   // Get component behaviour in all bMSCs
5    ForEach bMSC b in S.getbMSCs()
6      print c.name() + ``_'' + b.name() + `` = '' + getBehaviour(c, b);

8   // Link behaviour according to hMSC
9    print ``HMSC_'' + c.name()+ `` = '' +(`'' + getAdjacent(S.gethMSC().getInitialNode(),
         S.gethMSC()) + ``)'';

11  // first map nodes to behaviour and to their adjacent nodes
12   ForEach Node n in S.gethMSC().getNodes()
13     b = S.gethMSCs().map(n);
14     string nodes_behaviour = c.name() + ``_'' + b.name();
15     print n.id() + ``='' + nodes_behaviour + ``;'' + n.id() + ``_Adj'';

17  // second connect nodes according to hMSC arcs
18   ForEach Node n in S.gethMSC().getNodes()
```

```
19      print n.id() + ``_Adj) = (`` + getAdjacent(n, S.gethMSC()) + ``)'';

21   // Determinise, minimise and hide internal actions.
22    print ``deterministic `` + c.name() + `` = HMSC_``+c.name()+``/{intAction}.'';
23  }

25   // Extension: compose message label with its reliability and failure probability
26  String getBehaviour(Component c, bMSC b) {
27    String s, w, e, msg;
28    int next;
29    double rel, error;
30    Instance i = b.getInstance(c);
31    String label = c.name() +``_``+b.name();
32    for (int a = 0 ; a < i.size() ; a++) {
33      next = a+1;
34      e = ```';
35      msg = i.getEvent(a).label();
36      rel = i.getEvent(a).weight();
37      if (rel < 1.0)
38        error = 1 - rel;
39        e = ``\n | (``+ error +``) `` + msg + `` -> ERROR'';
40      w = label + ``_Q``+ p + `` = ( ( `` + rel + `` )`` msg+`` -> ``+ label + ``_Q``+
            next;
41      s = s +  + w + e + ``),'';
42    }
43    s = s + ``END'';
44    return s;
45  }

47  // Extension: compose continuations with transition probability between scenarios
48  String getAdjacent(Node n, hMSC H) {
49    String s;
50    double pts;
51    if (H.getAdjacents(n) = 0)
52      s = ``END'';
53    else ForEach Node m in H.getAdjacents(n){
54      pts = n.transition(m).value();
55      if (!first node transition)
56        s= s +``|'';
57      s = s + `` (`` + pts + ``)  internalAction -> `` + m.id();
58    }
59    return s;
60  }
```

In step 1, the algorithm builds the behaviour of the component while traversing each pBMSC of the scenario specification. This process is represented in lines 5 to 6 and in method `getBehaviour` from lines 26 to 42 of the algorithm in Listing 4.1. In method

`getBehaviour` our extension over Uchitel et al. represented in lines 35 to 38 in Listing 4.1 consists in adding weights for each transition $C_i\_S_j$ of a component C_i in a scenario S_j representing the event of an invocation of a service offered by C_i. Those weights derive from the probability of success of the event, i.e. P(E), from the pBMSC. An action from the same process (or a transition from the same state, in LTS terminology) is also added representing the probability of failure for that event, having the global ERROR state as the target.

The resulting pair of transitions forms a probabilistic choice, with the former transition having probability R_{C_i} and the latter transition having probability $1 - R_{C_i}$. This step is illustrated in Listing 4.2. For instance, lines 16 to 22 represent the behaviour of the component Control in pBMSC Analysis. Notice that only action *data* has a weight representing the probability of success for that action and the corresponding action transiting to ERROR.

Listing 4.2: Behaviour of Control per Scenario in FSP Code

```
1  Control_Initialise = Control_Initialise_Q0,
2  Control_Initialise_Q0 = (s_Control_Initialise -> Control_Initialise_Q1),
3  Control_Initialise_Q1 = (start -> Control_Initialise_Q2),
4  Control_Initialise_Q2 = END.
5  Control_Register = Control_Register_Q0,
6  Control_Register_Q0 = (s_Control_Register -> Control_Register_Q1),
7  Control_Register_Q1 =  END.
8  Control_End = Control_End_Q0,
9  Control_End_Q0 = (s_Control_End -> Control_End_Q1),
10 Control_End_Q1 = (control.sensor.shutdown -> Control_End_Q2),
11 Control_End_Q2 =  END.
12 Control_Terminate = Control_Terminate_Q0,
13 Control_Terminate_Q0 = (s_Control_Terminate -> Control_Terminate_Q1),
14 Control_Terminate_Q1 = (stop -> Control_Terminate_Q2),
15 Control_Terminate_Q2 =  END.
16 Control_Analysis = Control_Analysis_Q0,
17 Control_Analysis_Q0 = (s_Control_Analysis -> Control_Analysis_Q1),
18 Control_Analysis_Q1 = (query -> Control_Analysis_Q2),
19 Control_Analysis_Q2 = ( (0.95) data -> Control_Analysis_Q3
20                      | (0.05) data -> ERROR),
21 Control_Analysis_Q3 = (command -> Control_Analysis_Q4),
22 Control_Analysis_Q4 =  END.
```

In step 2, the algorithm builds an FSP process for the behaviour of the component defined by the pHMSC. There are two intermediate tasks in order to build that process: one is to model the mapping from pHMSC nodes to pBMSCs (lines 12 to 15 in Listing 4.1) and the second is to model possible adjacencies of nodes in the pHMSC (lines 18 and 19, including method `getAdjacent` in Listing 4.1). For example, if we consider that nodes Register and Initialize have node id 0 and 1, then the synthesis produces

`Control_N1 = Control_Initialise;Control_N1_Adj` (line 3 of the Listing 4.3), which then continues in line 8, which creates a transition to node id 0 from the current node 1.

Listing 4.3: Behaviour of Control from the pHMSC perspective

```
1  HMSC_Control = ( (1.0) internalAction -> Control_N1),
2  Control_N0 = Control_Register;Control_N0_Adj,
3  Control_N1 = Control_Initialise;Control_N1_Adj,
4  Control_N2 = Control_End;Control_N2_Adj,
5  Control_N3 = Control_Analysis;Control_N3_Adj,
6  Control_N4 = Control_Terminate;Control_N4_Adj,
7  Control_N0_Adj = ( (0.7) internalAction -> Control_N0 |  (0.2) internalAction ->
       Control_N3 |  (0.1) internalAction -> Control_N4),
8  Control_N1_Adj = ( (1.0) internalAction -> Control_N0),
9  Control_N2_Adj = (endAction -> END),
10 Control_N3_Adj = ( (1.0) internalAction -> Control_N0),
11 Control_N4_Adj = ( (0.5) internalAction -> Control_N1 |  (0.5) internalAction ->
       Control_N2).
```

Our extension to Uchitel et al. approach regarding this step 2 is essentially in the mapping of adjacent nodes. In method `getAdjacent`, the scenario transition probabilities PTS_{ij} are mapped to weights on the hidden transitions, referred as `intenalAction` in Listing 4.1, linking the C_i-S_j. Notice for instance, that Register, as node id 0, has transitions to three adjacent nodes: 0 (Register itself), 3 (Analysis), and 4 (Terminate). Each of those transitions are annotated with the probability of transition to each scenario according to the values annotated in the pHMSC for the Boiler.

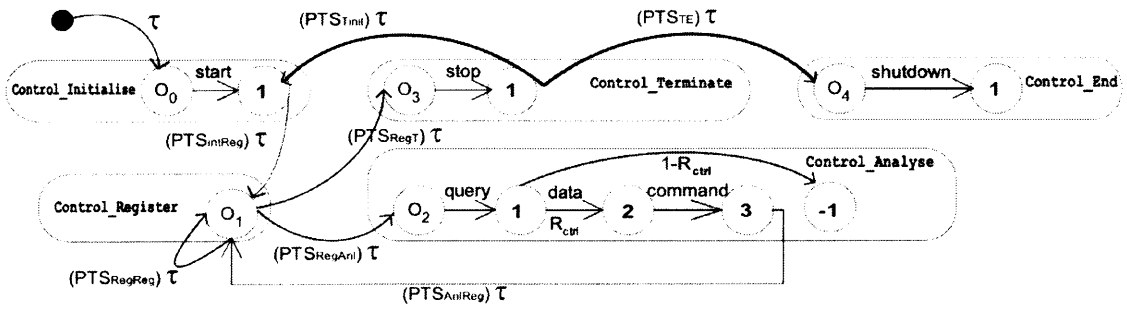


Figure 4.2: pLTS Synthesised for Component *Control*.

We illustrate as pLTS in Figure 4.2 those intermediate FSP processes in Listings 4.2 and 4.3 for the Control component. That pLTS for Control would be synthesised as the result of applying tasks 1 and 2 of our synthesis method. Each shaded area contains a pLTS synthesised in task 1 from a BMSC of Figure 4.1 and thus models the behaviour of *Control* within that BMSC. The transitions linking these different pLTS are synthesised in task 2 and correspond

to the transitions between BMSCs defined in the HMSC of Figure 4.1. Note that the weights on the τ transitions are the same as the corresponding transitions in the HMSC of Figure 4.1. Note also that because *data* is the only message received by *Control* in scenario *Analyse*, it is synthesised as two transitions, the "successful" transition being weighted with probability R_{ctrl} and the transition to the ERROR state (labelled -1 in the figure) being weighted with probability $1 - R_{ctrl}$. This action only applies to transitions labelled with *data*, following our assumption for the reliability of the invocation of a service to a component.

Continuing with our extensions, in task 3, the weights must be handled correctly in the process of reducing each component pLTS to first its minimal form (without internal τ transitions), and then to its deterministic form (without having outgoing transitions with the same label). In particular, Uchitel et al. , show the minimal form of the component preserves the component structure (instances of a BMSC) and interfaces (set of send and receive events) by means of equivalence relation resulting in a LTS with fewest states possible. Following their approach, we introduce some new tasks to accommodate the weights of the internal τ actions. The composition of probability follows Algorithm 1 for minimization.

Algorithm 1 Minimization of Component pLTS

Input: $\Gamma = (S, L, \Delta, q)$.

Output: $\Gamma' = (S', L', \Delta', q')$.

1: $\Gamma' \leftarrow \Gamma \backslash \{\tau\}$

2: $q' \leftarrow q$

3: $\Delta' \leftarrow \varnothing$

4: **for** each $s \in S$ **do**

5: do $\Delta' \leftarrow \Delta' \cup \{s, p, l, s'\}$, where $l \neq \tau$

6: **for** each $n \in Path(s)$ **do**

7: do $\Delta' \leftarrow \Delta' \cup \{(s, p \times p', l, u) : (n_i, p', l, u) \in \Delta, l \neq \tau$, for $0 \leq i < |n|\}$

8: **end for**

9: **end for**

where:

$Path(s)$ is a sequence of states $n = s_0, s_1, \ldots$ if $s_0 = s$, s $\in S$ where and $s_{i+1} \in Adj(s_i)$.

And $s' \in Adj(s)$ if $(s, p, l, s') \in \Delta$, where $\tau \in$ L.

Essentially, the algorithm for minimzation works through the elimination of a τ transition results in the merging of the transition's target state with its source state, with the outgoing transitions of the target state becoming outgoing transitions of the source state. Since there

Algorithm 2 Determinization of Component pLTS

Input: $\Gamma = (S, L, \Delta, q)$.

Output: $\Gamma' = (S', L', \Delta', q')$.

1: $\Delta' \leftarrow \varnothing$
2: $q' \leftarrow q$
3: **for** each $s \in S$ **do**
4: **for** each $s' \in Adj(s), (s, p, l, s') \in \Delta$ **do**
5: $Adj'(s) \leftarrow Adj(s)$
6: **if** $\exists s'' \in Adj'(s), (s, p', l', s'') \in \Delta, l = l')$ **then**
7: do
8: **for** each $s'' \in Adj(s'), s'' \neq s, (s', p'', l'', s'') \in \Delta$ **do**
9: $\Delta' \leftarrow \Delta' \cup \{(s', p \times p'', l'', s'') : (s'', p'', l'', i) \in \Delta\}$
10: **end for**
11: **for** each $s'' \in Adj(s), s' \neq s'', (s, p', l', s'') \in \Delta$ **do**
12: **if** l = l' **then**
13: do
14: **for** each $s''' \in Adj(s''), s''' \neq s, (s'', p'', l'', s''') \in \Delta$ **do**
15: $\Delta' \leftarrow \Delta' \cup \{(s', p' \times p'', l'', s''') : (s''', p'', l'', u) \in \Delta\}$
16: **end for**
17: $p \leftarrow p + p'$
18: **end if**
19: $\Delta' \leftarrow \Delta' \cup \{s, p, l, s'\}$
20: $Adj'(s) \leftarrow Adj'(s) - \{s''\}$
21: **end for**
22: **end if**
23: **end for**
24: **end for**

where: $s' \in Adj(s) if (s, p, l, s') \in \Delta$.

may be multiple τ transitions from the original source state (each with weight less than one), the weight of an eliminated τ transition must be "pushed" to the newly accumulated outgoing transitions, with the new weight on each such outgoing transition equal to its old weight times the weight on the eliminated τ transition.

Some 'filtering' needs to be done before running the algorithm: in case there are τ transitions to a stop or error state, or self-loop τ transitions, they need to be removed. However, in the presence of τ self-loops (such as the τ self-loop on state O_1 of *Control_Register* in Figure 4.2), it can be shown that such transitions can be eliminated entirely without any of the above merging or pushing of its weight. At the end of the elimination of outgoing τ transitions from a state, the weights on the outgoing transitions of the resulting state may not sum to one, in which case the weights must be normalised so that they do sum to one.

For the determinization process we devised the Algorithm 2. The algorithm consists in first obtaining all those transitions characterizing non-deterministic actions, i.e. all sets of outgoing transitions starting from the same state under the same action label and leading to different successor states. Each such set of non-deterministic transitions is merged to have the same successor state, with the probability weights of the original transitions summed to form the weight for the transition to the merged successor state. Additional care is needed to account for cycles in the state machine. In our work we manually followed that algorithm until Tan and Uchitel defined a more general algorithm in [TU06], which is a variation of the Mohri determinization algorithm [Moh97]. Tan and Uchitel then implemented their algorithm in LTSA, which we used for the case study described in Chapter 6.

Using the example parameters presented previously in Figure 4.1, the resulting minimised LTS for component *Control* is depicted in Figure 4.3 and the equivalently expressed as an FSP in Listing 4.4.

Listing 4.4: Minimized and Deterministic FSP process for the Control

```
Control = Q0,
Q0 = ( (1.0) start -> Q1),
Q1 = ( (0.667) query -> Q2
  | (0.333) stop -> Q4),
Q2 = ( (0.95) data -> Q3
  | (0.05) data -> ERROR),
Q3 = ( (1.0) command -> Q1),
Q4 = ( (0.5) shutdown -> Q5
  | (0.5) start -> Q1),
Q5 = ( (1.0) endAction -> Q6),
Q6 = END.
```

Finally, in task 4, we reuse the work from Ayles et al. [AFMB03] where the architecture model is constructed as the parallel composition of the minimal (without internal τ transition) and deterministic (no states with more than one outgoing transitions with the same label) pLTSs synthesized for each component. The weights of the pLTS for the architecture model are

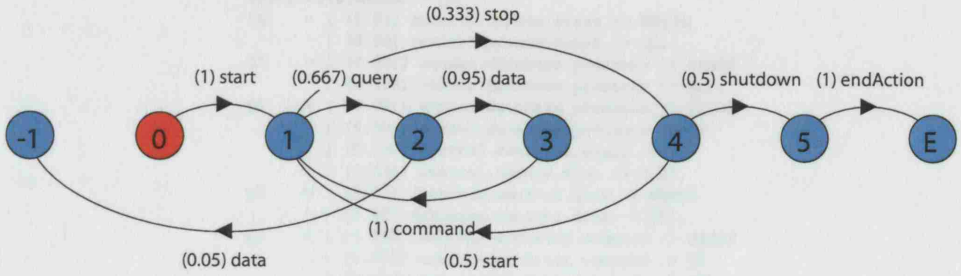


Figure 4.3: Minimised and Deterministic pLTS for Component *Control*.

computed according to the notion of *generative parallel composition* defined by D'Argenio et al. [DHK00].

Essentially, the resulting model is a pLTS where the sum of the discrete probabilities of all outgoing transitions for each state in a pLTS sum to 1. At the end of this task, it follows that for each node of the synthesized architecture model, $\sum_{j=1}^{n} PA_{ij} = 1$, where n is the number of states in the pLTS architecture model and PA_{ij} is the probability of transition between state S_i and S_j of the composed pLTS. Otherwise, $PA_{ij} = 0$ if the transition (S_i, S_j) does not exist.

The architecture model for the Boiler Control system resulting from the application of all four tasks of our extended synthesis method is depicted in Figure 4.4. For the sake of readability, we present the model in textual form as a specification expressed in FSP (Finite State Processes), the modelling notation of the LTSA tool (Labelled Transition System Analyser) [UCKM03]. FSP serves both as a modelling notation for end users, and as an intermediate form used in the automated synthesis of LTS models. As shown in the figure, a side-effect of the synthesis is the use of the auxiliary action *endAction* as the final action in a terminating path through the LTS.

4.2.3 Computing the Reliability Prediction

This final task consists in the prediction of software system reliability itself. In order to do so, we need a reliability model that extracts the system reliability from the probabilistic behaviour model. If we abstract away the labels in the transitions of the architecture model, a Discrete Time Markov Chain(DTMC) model is revealed.

> A DTMC model is a class of probabilistic automata where for *any* state the sum of the probabilities of all outgoing transitions equals 1.

The Cheung model [Che80] is an adequate approach for our goal. It computes the system reliability as a function of both the deterministic properties of the structure of the program (comparatively, the behaviour model) and the properties of components utilisation and failure following a DTMC model.

```
ArchitectureModel = Q0,
Q0    = ( (0.01) control.sensor.start -> ERROR
        | (0.99) control.sensor.start -> Q1),
Q1    = ( (0.001) sensor.database.pressure -> ERROR
        | (0.999) sensor.database.pressure -> Q2),
Q2    = ( (0.001) sensor.database.pressure -> ERROR
        | (0.809) sensor.database.pressure -> Q2
        | (0.152) control.database.query -> Q3
        | (0.038) control.sensor.stop -> Q10),
Q3    = ( (0.05) database.control.data -> ERROR
        | (0.95) database.control.data -> Q4),
Q4    = ( (0.005) control.actuator.command -> ERROR
        | (0.521) control.actuator.command -> Q5
        | (0.474) sensor.database.pressure -> Q9),
Q5    = ( (0.964) sensor.database.pressure -> Q2
        | (0.036) control.sensor.stop -> Q6),
Q6    = ( (0.005) control.sensor.start -> ERROR
        | (0.005) control.sensor.shutdown -> ERROR
        | (0.495) control.sensor.start -> Q1
        | (0.495) control.sensor.shutdown -> Q7),
Q7    = ( (1.0) endAction -> Q8),
Q8    = STOP,
Q9    = ( (0.006) control.actuator.command -> ERROR
        | (0.616) control.actuator.command -> Q2
        | (0.378) sensor.database.pressure -> Q9),
Q10   = ( (0.005) control.sensor.start -> ERROR
        | (0.005) control.sensor.shutdown -> ERROR
        | (0.495) control.sensor.shutdown -> Q7
        | (0.495) control.sensor.start -> Q11),
Q11   = ( (0.855) sensor.database.pressure -> Q2
        | (0.145) control.database.query -> Q12),
Q12   = ( (0.05) database.control.data -> ERROR
        | (0.95) database.control.data -> Q13),
Q13   = ( (0.005) control.actuator.command -> ERROR
        | (0.471) control.actuator.command -> Q1
        | (0.524) sensor.database.pressure -> Q9).
```

Figure 4.4: The FSP of the Architecture Model.

Applying Cheung model to our synthesized behaviour model, the transition weights are mapped into a square transition matrix M' whose row entries sum to one. This is used as the matrix M' described in chapter 2 on section 2.2.3.2, with $N = \{E, -1, 0, 1, ..., n-1\}$ the set of states in the synthesised LTS, E the terminal state of correct execution (corresponding to state C described in Section 2.2.3.2), -1 the terminal fault state (state F of Section 2.2.3.2), and $n-1$ the state from which a transition to state E is made upon action *endAction* (state N_n of Section 2.2.3.2). Note that the numeric state labels produced by LTSA may need to be renumbered so that the state leading to state E is the highest numbered state, as required by Cheung's model.

In Figure 4.5 we depict the transition matrix derived from the synthesised architecture model presented in Figure 4.4; note that this is actually the reduced matrix M, with the rows and columns for states E and -1 eliminated as in Section 2.2.3.2. Additionally, we point out for the fact that the rows in the sparse matrix in Figure 4.5 will sum to one if we add the architecture model transitions to the *ERROR* state.

Applying the Cheung model to that matrix, we predict that the reliability for the Boiler Control system as *Rel*$= 0.632 = 63.2\%$. This can be interpreted through the Rate Of Failure

$$\begin{pmatrix}
0 & 0.99 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0.999 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0.809 & 0.152 & 0 & 0 & 0 & 0 & 0 & 0 & 0.038 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0.95 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0.521 & 0 & 0 & 0 & 0.474 & 0 & 0 & 0 & 0 \\
0 & 0 & 0.964 & 0 & 0 & 0 & 0.036 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0.495 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.495 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.95 & 0 & 0 & 0 & 0 & 0 \\
0 & 0.471 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.524 & 0 & 0 & 0 & 0 \\
0 & 0 & 0.616 & 0 & 0 & 0 & 0 & 0 & 0 & 0.378 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.495 & 0.495 & 0 \\
0 & 0 & 0.855 & 0 & 0 & 0 & 0 & 0.145 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

Figure 4.5: The Matrix Derived from the Synthesized Boiler LTS.

Occurrence (ROCOF) [Som01] reliability metric. Therefore, out of a thousand runs, 368 (1000 minus 632) failures are likely to occur.

4.2.4 Implied Scenarios Detection in Reliability Analysis

In chapter 3, we presented the importance of taking implied scenarios detection into account in scenario based analysis models. In the example in Figure 4.1, we saw that the Boiler Control System showed one of the implied scenarios, depicted on Figure 3.4, where a query in the Database was being performed immediately after start, which is not in accordance with the sequence of messages specified in the BMSCs and the HMSC of the Boiler System.

The existence of an implied scenario means the Boiler Control System has been applied on a scenario specification that has a mismatch between behaviour and architecture. The behaviour model constructed in the previous section to predict reliability can exhibit behaviour (an implied scenairo) that has not yet been validated and that, according to whether it described intended or unintended system behaviour, can impact system reliability.

As an example, suppose that the rate at which the sensor checks pressure information and saves it in the database is high enough that the probability of occurence of the trace in Figure 3.4 is negligible. Then reliability should be predicted on the behaviour model of Figure 4.4 constrained in such a way that the implied scenario cannot occur, referred to as *negative implied scenario*. We can use the approach described in [UKM04] to build such a constraint, depicted in Listing 4.5.

Listing 4.5: The Implied Scenario Constraint FSP

```
NegScen = (control.sensor.start -> Aux
      | {sensor.database.pressure, control.database.query} -> NegScen),
Aux = (control.sensor.start -> Aux | sensor.database.pressure -> NegScen).
```

Notice the FSP for the negative implied scenario detected is not annotated with weights for the transitions. Considering those traces in the implied scenarios are not occurring in the synthesised behaviour model, it is a natural choice not to annotate them. On the other hand, if the traces in that scenario were wanted ones, they need to be incorporated into the scenario specification and become a pBMSC node of the pHMSC following tasks one to four from our approach in order to predict the reliability of the new specification.

After the constraint is built so the detected implied scenario cannot occur, the new constrained architecture model is the result of the parallel composition between the previous behaviour model and implied scenario constraint, as presented below:

```
||ConstrainedArchitectureModel = (ArchitectureModel || NegScen).
```

If we calculate the reliability of the resulting constrained model in the same way as described in Section 4.2 then we obtain 86.2%. Similarly to the previous result, we can interpret this result through the ROCOF reliability metric where, out of a thousand runs, 138 (1000 minus 862) failures are likely to occur.

Either positive or negative, implied scenarios can impact the reliability prediction significantly and for that, they should be validated before reliability is calculated.

More generally, the existence of implied scenarios as a result of the close relation that exists between behaviour and architecture in scenario-based specifications supports our claim that taking into account behaviour and architecture when performing reliability prediction is important.

4.2.5 Performing the Sensitivity Analysis

The sensitivity analysis we carry out using our technique consists in determining how the system reliability varies as a function of the components' reliabilities and scenario transition probabilities, with the purpose of identifying probabilities that have the greatest impact on the reliability of the software system [RRU05b]. In this section we illustrate some sensitivity analyses that can be performed for our reliability prediction technique. We carry out two different empirical analyses of our technique: (1) as a function of the components' reliability and (2) as a function of the transition probability between scenarios. The analyses are exemplified using the Boiler Control system.

4.2.5.1 System Reliability as a Function of Component Reliability

This analysis consists in varying the system reliability as a function of the components' reliabilities with the purpose of identifying components that have the greatest impact on the reliability of the software system. The method consists of varying the reliability of one component at a time and fixing the others to 1. The transition probability values are those for PTS_{ij} presented in Figure 4.1, where i and j represent respectively the *from* and *to* scenarios of the transition. Figure 4.6 shows the graphs of the reliability of the system architecture model as a function of

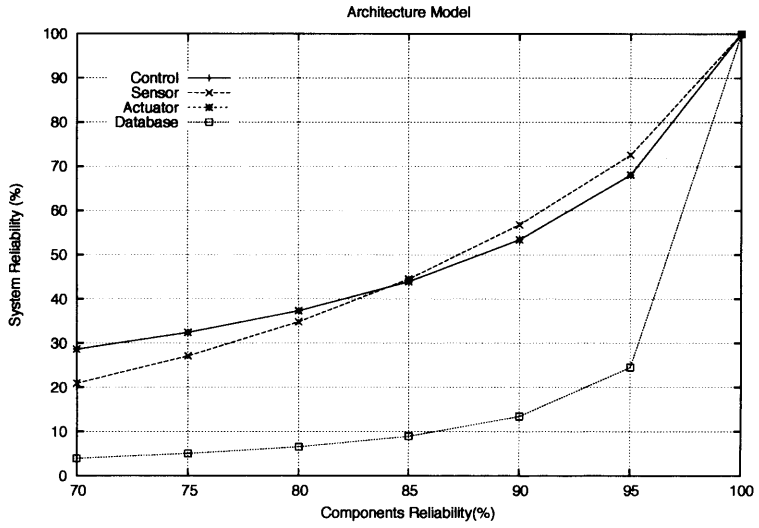


Figure 4.6: The System Reliability of the Architecture Model as a Function of the Component Reliabilities.

the component reliability. Note that the component *Database* has a large impact on the system reliability, in such a way that the system reliability drops quickly *Database* reliability decreases below 100%. We can initially attribute this to the fact that the system reliability is directly proportional to the number of requests that each component processes. But not only, otherwise *Sensor* would have the strongest impact on the system reliability.

We consider this result due to the higher probability of transitions to scenarios where *Database* takes part than to scenarios where *Sensor* takes part. The *Database* is executed in scenarios *Register* and scenario *Analyse*. According to the Figure 4.1, the outgoing transitions of scenario *Register* have chances of following three different paths: (1) 70% chance that scenario *Register* will execute again, (2) 20% chance that scenario *Analyse* will be executed and (3) 10% chance that scenario *Terminate* will be executed. This gives component *Database* 100% chance that it is executed, considering successful the transition from scenario *Initialise*

to scenario *Register*, followed by 70% chance from the execution of the *Register* loop transition and another 20% chance of transition from scenario *Register* to scenario *Analyse*. On the other hand, the *Sensor* participates in scenarios *Initialise*, *Terminate* and *End*. Chances that the *Sensor* will be invoked correspond to 100% for *Initialise* execution, 10% for the execution of scenario *Terminate* and 50% for the execution of scenario *End*. Compared to the results for the *Database*, we can figure out why *Database* has a higher impact on the reliability of the Boiler compared to the *Sensor*.

Components *Control* and *Actuator* identically show less impact on the system reliability. This is because they are always invoked in the same scenario, *Analyse*, the same number of times, once. Therefore, they are expected to equally have less impact on the overall system reliability.

4.2.5.2 System Reliability as a Function of Transition Probability

This analysis consists in varying the system reliability as a function of the scenario transition probabilities. Considering scenarios with multiple outgoing transitions, we want to find out if scenario transition probabilities have a significant influence on our reliability predictions. In case that influence is significant, we want to find out which of those transitions has higher impact on system reliability. Using the Boiler system as an example, we analyse the impact of outgoing transitions from scenarios *Register* and *Terminate*, as the other scenarios have only one outgoing transition with unitary probability transition. To run this experiment, we keep the component reliabilities values in Figure 4.1. Taking one transition of scenario *Register* and scenario *Terminate* is enough to analyse the sensitivity of the system reliability as a function of the transition probability. This is due to the fact that outgoing transitions of a scenario sum to a unity. Varying one transition, we vary the remaining outgoing transitions of the same scenario proportionally, so that transitions sum to 1. For instance, consider the outgoing transitions of scenario *Register* in Figure 4.1. If we change PTS_{RegReg} from 0.7 to 0.1, we allocate the remaining 0.9 to PTS_{RegAna} and PTS_{RegTer} in a way that preserves the ratio between them (0.6 and 0.3, respectively). For further information on probabilistic composition of concurrent processes, we refer the reader to D'Argenio et.al [DHK00].

The results depicted in Figure 4.7 show that outgoing transitions from scenario *Terminate* have more impact on the overall system reliability than outgoing transitions from scenario *Register*. This impact is a result of the role that scenario *Terminate* plays in the whole Boiler system. Intuitively, this can be reasoned by the fact that the higher the probability of transition from *Terminate* to *Initialise*, the higher the chance that the whole system will fail, as both scenarios comprise the end and the beginning of an iteration through the system. Conversely,

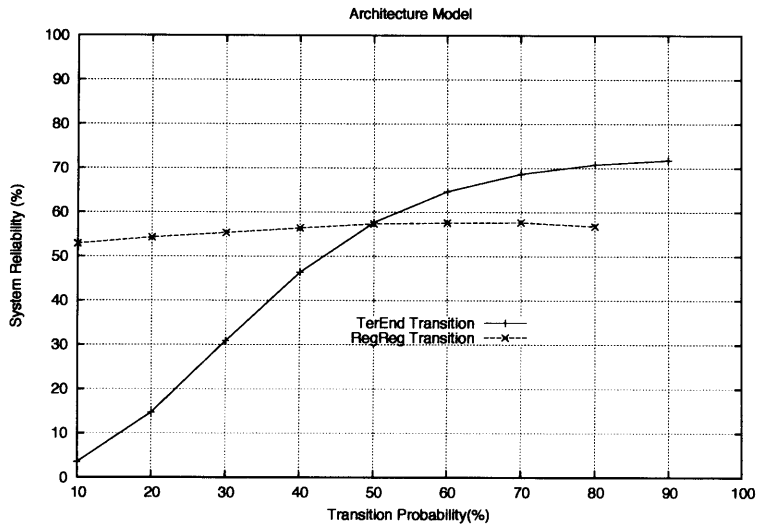


Figure 4.7: The System Reliability of the Architecture Model as a Function of the Transition Probabilities

the higher the chance the transition from *Terminate* to *End*, the higher the system reliability, meaning fewer loops will happen from *Terminate* to *Initialise* and therefore fewer chances for the failure of the system.

4.2.5.3 Comparing the Reliability Prediction Curves

In previous sections, we compute reliability predictions in the infinite extreme over all possible executions, as provided by Cheung model [Che80] and our previous work [RRU05c]. In this section, we compute reliability predictions as a function of the number of scenario executions.

To measure the system reliability after a certain number of system executions, we start from the Cheung definition that the system failure probability, E, is the probability of reaching state F (faulty termination) from the initial state N_1:

$$E = P^n(N_1, F) \tag{4.1}$$

P represents the transition probability matrix with all the state transitions of the synthesized probabilistic architecture model, including the transitions to the absorbing states C (correct termination) or F, (faulty termination). $P^n(i, j)$ is the probability that starting from state i, the chain reaches state j at or before the n^{th} step.

As a result of Equation 4.1, we have a matrix that, after n steps, results in the probability that execution traverses from state N_1 to the final faulty state F. In order to obtain the graphs in Figure 4.8 we follow three steps: (1) Iterate P n times: P^n; (2) Obtain from P^n the probability

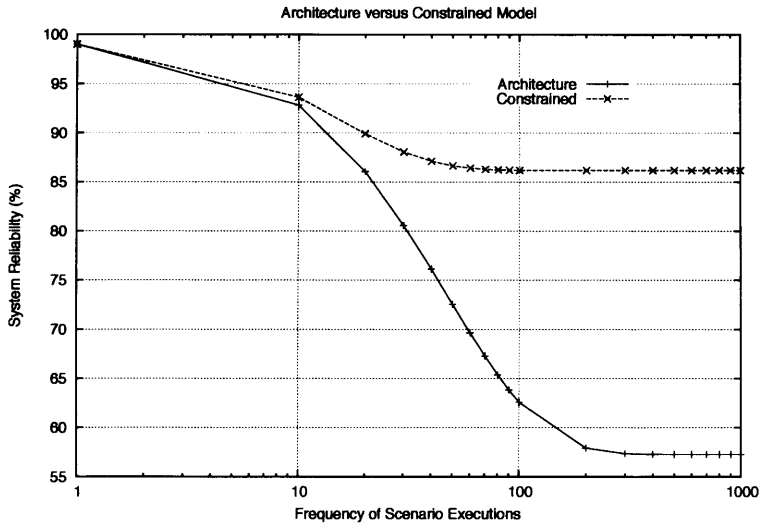


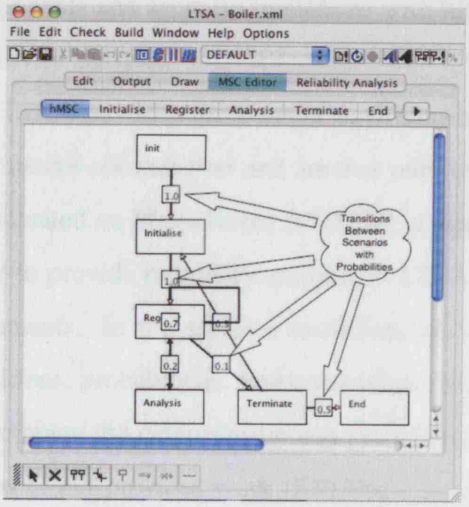
Figure 4.8: The System Reliability as a Function of the Frequency of Scenario Executions.

of failure of the system E: the probability of reaching the absorbing fault state F from the initial state N_1; (3) Calculate the reliability of the system as $R = 1 - E$.

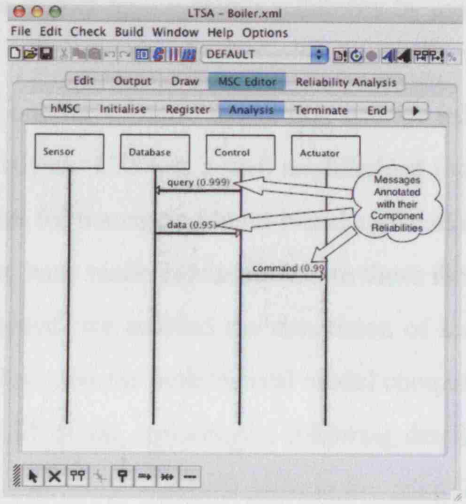
The analysis of Figure 4.8 shows that the greater the number of scenarios executed, the more noticeable is the difference between the reliability predicted for the Architecture Model and the Constrained Model. Furthermore, the reliability flattens out at around 300 scenario executions for the Architecture Model and 70 for the Constrained Model, as the likelihood of avoiding the *End* scenario becomes minuscule after those points.

4.3 Implementation

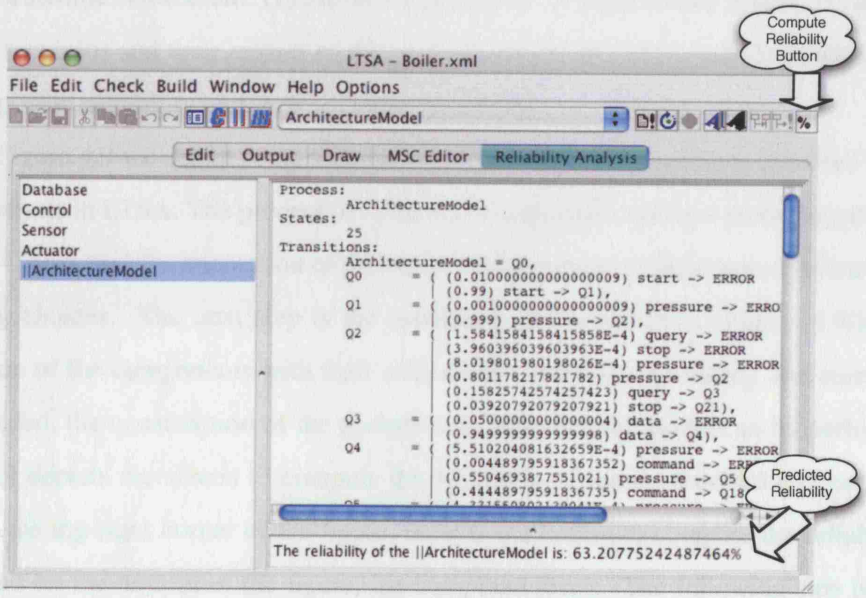
We implemented a plugin in LTSA to automate the reliability analysis using our prediction technique. LTSA allows the use of behavioural models of distributed systems as prototypes for exploring system behaviour, and for automated checking of model compliance to properties (i.e., model checking). To support reliability prediction, our new extensions allow annotating a scenario specification with probabilities and using LTSA to process the resulting scenarios. The enhanced LTSA tool then compiles these annotated scenarios by applying probabilistic Finite State Process (FSP) generation. We implement Cheung model in our plugin for LTSA to estimate the reliability from the generated probabilistic architectural model. It is important to note that the minimization and determinization of the synthesized FSP is a contribution of Tan and Uchitel who implemented their algorithm in LTSA according to their work in citeChuanUchitel:2006.



(a) The Boiler pHMSC in LTSA



(b) A Boiler pBMSC in LTSA



(c) The Screen to Compute Reliability

Figure 4.9: Reliability Computation Using LTSA

The advantages of using LTSA for reliability analysis are twofold. Firstly, the tool enables the system stakeholder to model the system closer to their own perspective instead of requiring the users to provide very fine grained state machine models of the system. Secondly, it provides the ability of identifying interactions that arise from the concurrent nature of the system's behaviour, the so-called *implied scenarios* [UKM04], which can potentially decrease the system's reliability. These two benefits together make LTSA a suitable tool for reliability analysis. However, the purpose of the tool is not to model all the component interactions of a system. The main aim is to provide a bird's eye view of the system focusing on reliability, and to identify

components and scenario transitions with higher impact on a system's reliability by means of sensitivity analysis.

LTSA provides three major environments: one for scenario modelling, one for architectural model computation and another one for analysis. LTSA is a well modularized tool, implementated on MagicBeans [CEM04], a platform for managing plugin-based applications. In order to provide reliability analysis in LTSA, we have made enhancements to these three environments. In the scenario modelling environment, we enabled the annotation of scenario transitions, probabilities, and component reliabilities. In the architectural model computation, we compute the deterministic and probabilistic LTS of the components, following details presented in our previous work [RRU05c]. Finally, the analysis results provide the reliability of the system by applying the Cheung computation over the synthesized architectural model. The analysis outcome is threefold: (1) reliability prediction of the software system (1) elicitation of implied scenarios and their impact on the overall system reliability, and (2) sensitivity analysis for component reliability and their scenario transitions.

In Figure 4.9 we depict some screenshots of the major environments involved in the reliability analysis in LTSA. The process of reliability computation starts at modelling the pHMSCs (Figure 4.9(a)), and the annotation of probability information on those scenario transitions representing choices. The next step is the modelling of the pBMSCs (Figure 4.9(b)), and the annotation of the components with their reliabilities. After the modelling and annotation step is concluded, the construction of the probabilistic architectural model can be performed. Figure 4.9(c) depicts the screen to compute the reliability of the synthesized architecture model FSP. On the top right corner of the figure, there is the button to compute the reliability of the model and on the bottom of the figure, the computed result. The following step is to use the reliability plugin to compute the system reliability from the synthesized architectural model. Future enhancements to the implementation includes fully automated sensitivity analysis and importing the visualization of the sensitivity analysis into the reliability analysis plugin.

The implementation still need some improvements though. At the moment, few inconveniences in the implementation of the deterministic algorithm have been reported, although they do not impede the computation of the reliability per se. An improvement of the implementation for the future is to remove an extra STOP state, which was introduced in the process of making deterministic the component LTS. However, the differences between number of states of manual and automated determinization are negligible.

The other improvement that needs to be done is related to the sensitivity analysis which is not fully automated, where at the moment, the range of input values are performed manually.

Future versions of the plugin will act on those issues.

4.4 Related Work

Several previous architecture-based approaches to reliability engineering of component-based systems have been reported. We introduced some of them on chapter 2, but Goševa-Popstojanova and Trivedi provide a more comprehensive survey of the various approaches [GPT01]. We revisit some of those architecture-based models, mainly the state- and the path-based ones, from the perspective where they can be compared with our technique.

State-based models [Che80, GLT98] use a control flow graph to represent the system architecture. In such models it is assumed that the transfer of control among the components can be modelled as a Markov chain, with future behaviour of the system dependent only on the current state and not on past behaviour. Gokhale et al. use a regression test suite to experimentally determine the architecture of the software and the reliabilities of its components. As described in chapter 2, Cheung's model takes into account the reliability of each component and the operational profile. In general, relying the analysis of the software reliability on state-machines provided by stakeholders only, may not be accurate. In our model, the system states are generated by the LTSA based on the precision of a model checker. Although scenarios are provided as a basis for the analysis, we explore the expressiveness of the given scenarios by checking if the existence of implied scenarios that could impact negatively during the system execution. Reussner etal. [RSP03] also uses Cheung model to compute reliability of component based systems based on their parameterized contracts. According to Reussner et al., those contracts are based on mappings between provided and required gates of the architecture model of a software system. The major difference between their work and ours is that they do not consider the concurrent nature of the components when composing the set of states of the software architecture.

Path-based models [Sho76, YCA99] compute the reliability of the system by enumerating possible execution paths of the program. The scenario-based method of Yacoub et al. [YCA99] is perhaps closest in spirit to our own approach. In many ways their method is a hybrid approach in which a state-based model of the system is constructed from a scenario specification (a set of basic scenarios plus a graph representing the composition of basic scenarios), and then paths through the model are enumerated until a threshold execution time is reached along each path. Their approach reveals the pitfalls of using imprecise, coarse-grained behaviour models of system architecture. The model used in their approach is the *component dependence graph* (CDG), a state-machine model in which the states represent execution inside a partic-

ular component (with one state per component), and the transitions represent the transfer of control from one component to another (with a transition from one component to another representing a merge of all messages sent by the former to the latter in the scenarios). Because the representation of component behaviour in the CDG is at the level of whole components, it is an inherently sequential model of system behavior in which one component executes at a time, meaning that any concurrency inherent in the scenario specification is lost. Furthermore, a CDG can exhibit sequences of component transitions not found in the scenarios from which it is derived. In a sense such sequences are implied scenarios, but they arise not as an artefact of components having limited local knowledge of global behaviour. Instead, they are merely a consequence of modelling the system architecture imprecisely at the granularity of whole components rather than at the granularity of the component interactions specified in the scenarios. Finally, it can happen that a component in a CDG is represented by an absorbing state, even though the scenario specification itself is able to progress beyond any interactions with the "absorbing" component. Indeed, we attempted to model the Boiler Control system using the approach of Yacoub et al., with the result that the *Actuator* was an absorbing component from which we had to add transitions artificially to other components in order to construct a model that was able to progress to the final state.

Other work can be situated in the area of a model-driven analysis technique: [MH02, MPB03, CSC02]. These approaches also propose a framework for automatic generation of reliability models from software specifications, bringing reliability analysis to early stages of the software lifecycle. István et al. [MH02, MPB03] shed some light on ways to fully automate dependability analysis, applied to the Fault-Tolerant CORBA, using graph transformations into their VIATRA framework. The work from Singh et al. [SCC+01] provides a prediction algorithm to analyse the reliability of the system prior to its construction. Their approach requires the user to provide global behavior scenarios other than the local behavior of the components interactions. However, this feature may turn out to be unsuitable for the system modularity and therefore hindering systems maintainability.

4.5 Summary

In this chapter, we described a technique to predict software system reliability as a function of component reliability estimates. The technique consists on annotating a scenario specification with probabilistic properties and using a probabilistic labelled transition system (LTS) synthesised from the scenario specification for the software reliability prediction. We systematically structured the technique into five major tasks: (1) annotation of the scenarios, (2) synthesis of

the probabilistic LTS, (3) construction of the transition probability matrix, (4) system reliability prediction, and (5) implied scenario detection. As a result, we could analyse the software for its sensitivity of the components' reliabilities and probability of transition between scenarios, when more than one scenario can be executed at the same time. We also verified how implied scenarios analysis can impact considerably the reliability results. Following the tasks to accomplish our approach for model-driven software reliability prediction, we introduced the tasks delimited in the highlighted box of Figure 4.10.

In the next chapter, we devise a UML profile to encompass reliability from system design to its deployment. Due to the various nature of software systems and the need to have appropriate reliability metrics for each of those, the purpose of the contribution presented in this chapter is not meant to be the only reference of its kind. However, it adheres to our purpose towards accomplishing a comprehensive reliability profile for model driven engineering.

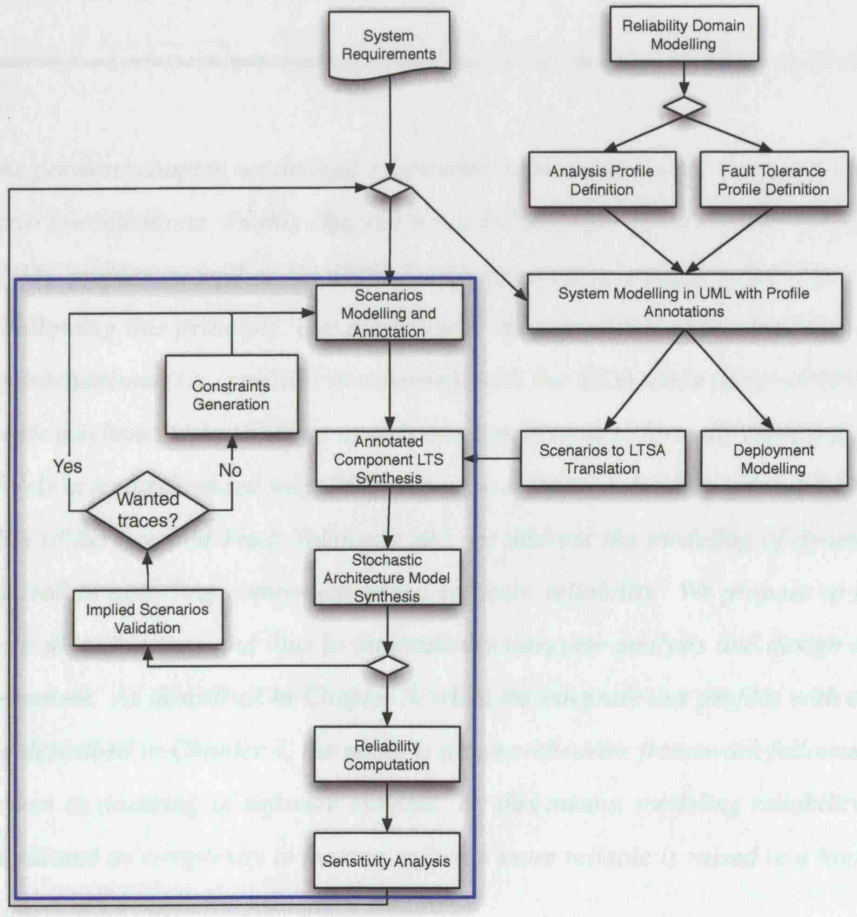


Figure 4.10: Highlighted Elements of Model Driven Reliability Prediction Framework Discussed in Chapter 4

Chapter 5

Supporting Reliability Engineering in the UML and MDA

In the previous chapter, we defined a technique to predict software system reliability based on scenario specifications. In this chapter, we define a profile for reliability modelling by extending UML profiles as well as the UML 2.0 specification to support reliability analysis and design. Following this principle, our approach to meta-modeling using the UML lightweight extension mechanisms (i.e., profiles) is consistent with the MDA white paper [OMG01], which defines basic mechanisms to structure models consistently and to formally express the semantics of the models in a standardised way. With respect to software reliability, previous UML profiles for Quality of Service and Fault Tolerance did not address the modeling of dynamic aspects often required in modeling component-based software reliability. We propose to amend that profile with those features and thus to semantically integrate analysis and design models into one environment. As described in Chapter 3, when we integrate our profiles with the analysis technique described in Chapter 4, the result is a comprehensive framework following standard model driven engineering of software systems. By this means, modeling reliability is considerably facilitated as complexity in making software more reliable is raised to a higher level of abstraction.

With the lack of an adequate reference model for reliability in the current MDA, we propose to tackle this problem in the levels of abstraction suggested by OMG's MDA, depicted in Figure 2.1. It is feasible to accomplish this task using the standard meta-modeling approach of MDA and specifications. Such a task has been demonstrated in the context of performance and real-time specification [OMG05]. Thus, in this chapter we present profiles we devised to support reliability engineering for component-based system in Model Driven Architecture.

The presentation of our contribution for this chapter follows the approach we started in Chapter 3, focusing on the steps depicted in Figure 5.1. In Section 5.1 we define the rationale behind the structuring of the reliability modelling framework where we specify the profiles for analysis, deployment and fault tolerance integrated through well-formedness rules. In Sections 5.2, 5.3, 5.4 we develop the reliability modelling framework. In particular, in Section 5.4 we also define a platform specific model for fault tolerance in EJB based on experience built on a well known EJB vendor. Finally, in Section 5.5 we compare our MDA-based reliability engineering process to other work related to our contribution to this chapter.

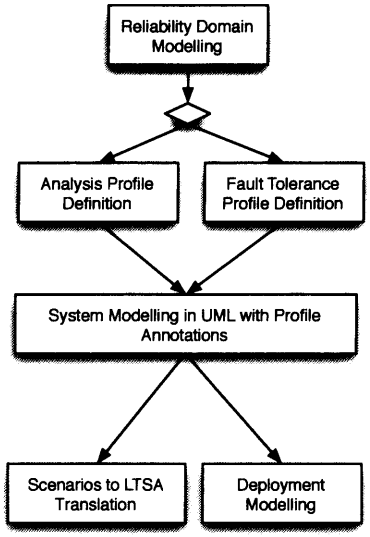


Figure 5.1: The Model for Reliability Prediction in MDA.

Note that we present no evaluation of these profiles per se. However, through examples we show these profiles are rooted on acknowledged good principles of engineering software systems. Therefore, we argue that these profiles simply incorporate a comprehensive set of features that any UML profile designer would need to include so as to support component-based reliability engineering in the MDA.

5.1 The Rationale for Reliability Modeling

It is well known in the software engineering community the interest in closing the gap between design and quantitative evaluation of software systems. However, the greater the demand on integrating design and analysis, the greater the need to provide software engineers, researchers and practitioners with the ability to manage software quality throughout its lifecycle; in particular, during its early stages, for the well known reasons of saving costs, time and efforts. This is the rationale behind a great deal of interest in combining quantitative analysis with model driven engineering.

As a result, providing the means to increase the ability to manage software quality has been one of the key focuses of the MDA community. Software reliability is one of the major software quality attributes for quantitatively expressing the ability of a system to deliver its services. In order to contribute a way to provide a systematic and consistent integration between system design and reliability analysis we follow standard MDA approach.

To be in accordance with the OMG standards, we extend the SPT profile [OMG05], in particular the General Resource Modelling(GRM) as it provides a uniform basis for attaching quantitative information of quality of service to UML models. As stated in chapter 3 of the SPT Profile Specification [OMG05]:

> The (GRM) framework captures QoS information represented either directly or indirectly, the physical properties of the hardware and software environments of the application (...) The GRM is envisaged as the foundation required for any quantitative analysis of UML models.

The GRM is organized into two related viewpoints: the *domain viewpoint* and the *UML viewpoint*. The domain viewpoint captures behavioural concepts, common structures and pattern that characterize the real-time system and its analyses methods. The domain viewpoint is defined independently of the UML metamodel. Whereas the UML viewpoint specifies how the elements in the domain model are realized in UML by making use of UML extension mechanisms, i.e. stereotypes, tag definitions and constraints.

We follow the organization of the GRM in order to build an MDA-based framework for modelling reliability as a quality of service. And more importantly, we create the reliability modelling framework based on the GRM framework and thus, following the OMG standard for modelling QoS properties.

In Figure 5.2 we relate the packages we propose for reliability modelling to those packages of the GRM framework. In the center of the GRM there is the Core Resource Model (CRM)

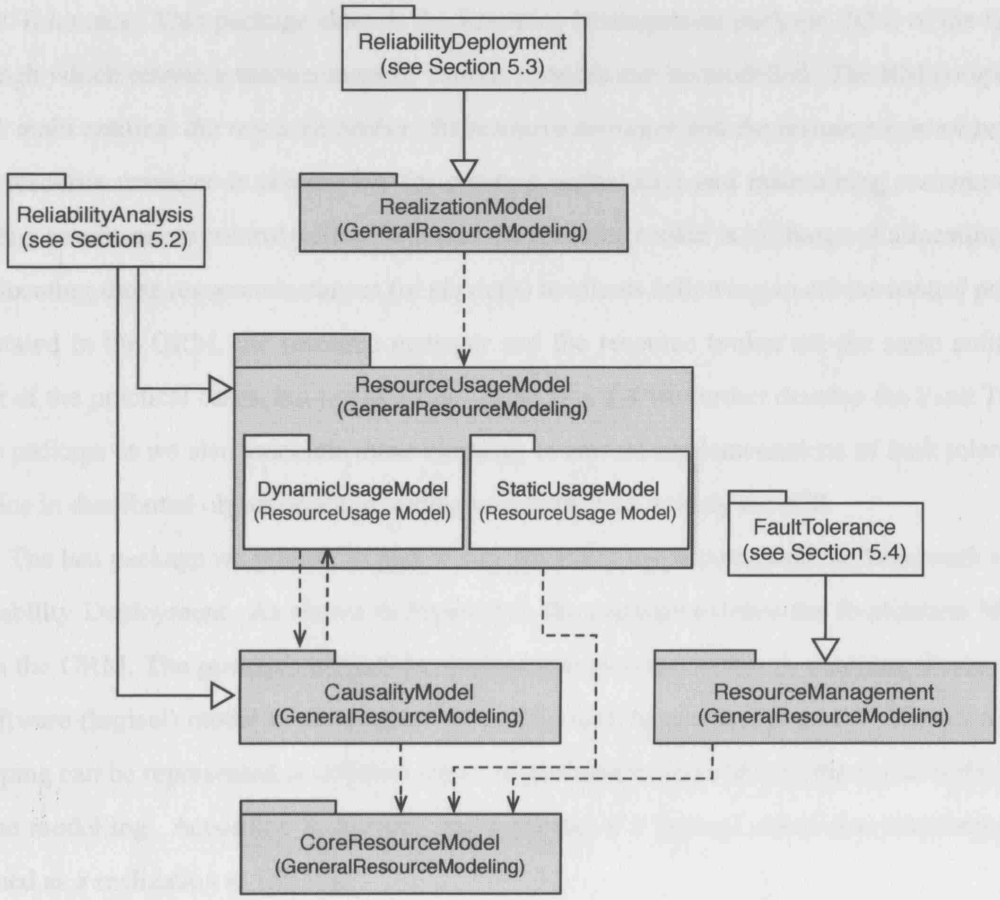


Figure 5.2: The Reliability Resource Modelling and the GRM Framework.

package. The CRM defines *design-time descriptor* and *run-time instance* elements with the essential concepts of resources and quality of service. It serves as a foundation for other types of QoS domains, in our case, reliability. As a result, it constitutes the basis that all the other packages we specify in this chapter are built on.

The Reliability Analysis package is intended for modelling the elements required on dynamic analysis of reliability as a quality of service. The importance of capturing the dynamic aspects related to reliability modelling is that software failures are revealed on the component interactions according to the scenarios of system execution. Such importance can be noticed, for instance, by the reliability analysis technique we defined in Chapter 4. For this reason, the analysis of reliability we focus on is for its dynamic usage. The GRM packages related to dynamics of QoS analysis are the Causality and the Dynamic Resource Usage models. We extend those models and delve into more details on our proposed reliability analysis package in Section 5.2. We also provide an example of usage for our analysis package by applying it to the Boiler Control system in that section.

Following the structure of the reliability resource modelling, we also define a package for

Fault Tolerance. This package extends the Resource Management package (RM) of the GRM through which resource management of various services can be modelled. The RM comprises three main entities: *the resource broker*, *the resource manager* and *the resource control policy*. The resource manager is responsible for creating, initializing and maintaining resources according to a resource control policy. Whereas the resource broker is in charge of allocating and deallocating those resource instances (or services) to clients following an access control policy. As stated in the GRM, the resource manager and the resource broker are the same entity in most of the practical cases, but not as a rule. In Section 5.4 we further develop the Fault Tolerance package as we also associate those elements to current implementations of fault tolerance service in distributed object oriented middleware platforms, mostly the EJB.

The last package we present as part of our reliability resource modelling framework is the Reliability Deployment. As shown in Figure 5.2, the package extends the Realization Model from the GRM. The principle behind the deployment package is that of mapping elements of a software (logical) model to elements of an environment (engineering) model. Therefore, the mapping can be represented at different levels of abstraction according to the needs at the time of the modelling. According to the SPT, the mapping of a general abstraction relationship is defined as a realization as follows:

> [A relationship that relates] a specification model element or elements (the supplier) and a model element or elements that implement it (the client).

In the realization mapping, the supplier is on higher level of abstraction compared to the client. Both clients and supplier can be a logical element, but only clients can represent an element in the engineering model. We extend the realization mappings for our reliability deployment model as a specialization of the *RealizationModel* package in Section 5.3. We also create constraints in the Deployment package which consistently models required and offered level of reliability so that fault tolerance is achieved with the appropriate number of replicas, presented in Section 5.4.1.

5.2 The Reliability Analysis Package

We believe the QoS Profile is not comprehensive enough to support reliability analysis, as it does not address the modeling of dynamic aspects (such as scenarios, component interactions, and operational profiles) often required in modeling component-based software reliability. On the other hand, dynamic aspects have been defined in the SPT (Schedulability, Performance and Real-Time) Profile but they were not incorporated into the QoS Profile. In this section, we define a UML profile for software reliability analysis and evolution of its reliability predictions.

First we define the domain model with the building block elements for an architecture based reliability analysis. Following the presentation style adopted in the SPT, we then define a UML profile to represent that reliability model. We also demonstrate how reliability analysis can be integrated into UML models having the reliability prediction technique we presented in previous chapter as an the instance model .

5.2.1 The Reliability Prediction Domain

A sub-profile for reliability analysis should support modelling tools for various kinds of reliability models. However, considering the vast amount of reliability analysis models already created, we focus on reliability prediction models that can support early reliability analysis through scenario specifications for the various benefits they can provide, such as efficiently capturing system requirements. Additionally, we want to capture the execution runs of systems where their reliability can be derived with regard to the system's usage scenario. As mentioned in previous chapters, it is identifying system failures (and not faults) that matter when one wants to analyse and improve the reliability of a system. And system failures are revealed when systems are in execution. For this reason, it is our interest to model the dynamics of the system in order to conceive a reliability analysis modelling in accordance with model driven standards.

For this reason, in order to accomplish the sub-profile for reliability analysis, we define two domains for reliability modelling: one generic following the structure of the Resource Usage Model package and a more specific one which models reliability prediction technique based on scenario specifications. We relate the two of them in section 5.2.2 where we define the UML viewpoint for the MDA-based reliability modelling.

5.2.1.1 A Generic Reliability Analysis Domain

The Resource Usage Model of the SPT can represent either the dynamic or the static usage of resources, according to the needs of the analysis. This analysis is defined in the Resource Usage package as the *analysis context* in order to assist model analysis tools in determining what part of a model is to be analyzed. The context consists of a set of resource usages and a set of resource instances used by the resource usages. We specify the analysis context from the Resource Usage Model as the *Reliability Context* in Figure 5.3, fully consistent with the Resource Usage framework in the SPT. The relationship of the general reliability modelling concepts to the Resource Usage package is also depicted in Figure 5.3.

In the static usage of a resource the relationship between clients and resources is viewed as static, while in the dynamic usage, the order and the time of occurrence of the loads on resources is relevant to the model analysis. In the core of the dynamic usage model there is

the *Scenario* which represents an ordered series of steps called *action executions*. The Scenario element is defined in the Causality Model which constitutes the basis for any dynamic modelling associated with the SPT profile.

Our interest on the analysis of reliability is for its dynamic usage. As a result, the reliability analysis package is an extension of the dynamic usage and Causality models. We model the resource usage model elements in Figure 5.3 in order to present the model from the perspective of reliability analysis. The dynamic elements in the reliability context are the same as those in the Dynamic Resource Usage model, i.e. the *Scenario* and *ActionExecution*, which may also explicitly specify the required QoS that it needs in order to meet its own obligations. The *RelResourceInstance* represents an abstraction of passive resource instance (*RelPassiveResource*) or active resource instance (*RelActiveResource*) resource participating in the scenarios.

In the Resource Usage Model, the event occurrence that causes the resource usage is called *Usage Demand*. The QoS characteristics associated with the usage represent the required QoS values of the system for that specific usage. In the context of reliability, the usage demand imposed on the system is expressed in terms of the *Service Delivery*, following Figure 5.3.

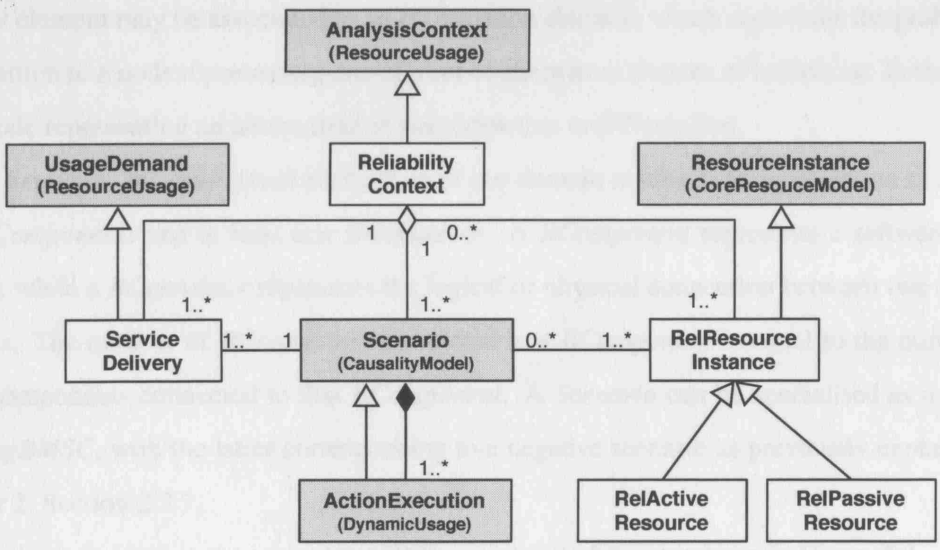


Figure 5.3: The Reliability Analysis Context.

5.2.1.2 A Specialized Reliability Analysis Domain

We specialize this reliability analysis context based on our reliability prediction technique presented in Chapter 4. The rationale behind our prediction technique is that the reliability of the system depends on two key pieces of information, as explained in Chapter 4: (1) scenario transition probabilities and (2) the reliability of the components. In order to support this approach within the MDA, we define the conceptual model of reliability prediction depicted in Figure 5.4.

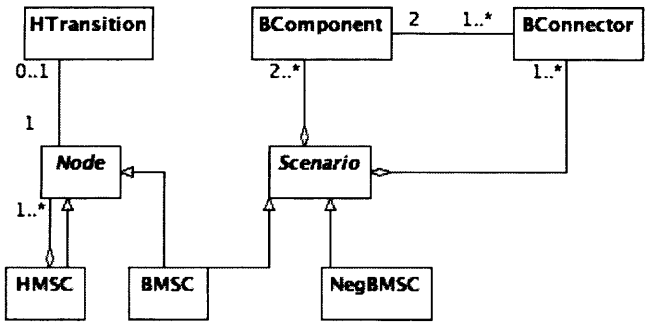


Figure 5.4: The Domain Model of the Reliability Prediction Technique.

There are two main abstract constructs in our domain model: the *Node* and the *Scenario*. A *Node* represents the nodes of an *HMSC*. These nodes can be specialized as a *BMSC* or another *HMSC*, in case of a hierarchical HMSC. A *BMSC* corresponds to a Basic Message Sequence Chart describing the interactions between components participating in a scenario, and an *HMSC* corresponds to the high-level structure representing the composition of *BMSCs*. A *Node* element may be associated to an *HTransition* element, which represents the probability of transition to a node representing one of a set of alternative choices of behaviour. In that case, each node representing an alternatives is stereotyped as an*HTransition*.

A *Scenario*, the other main abstraction of our domain model, is an aggregation of at least two *BComponents* and at least one *BConnector*. A *BComponent* represents a software component, while a *BConnector* represents the logical or physical connection between two *BComponents*. The number of *BConnectors* associated to a *BComponent* is equal to the number of other components connected to that *BComponent*. A *Scenario* can be specialised as a *BMSC* or a *NegBMSC*, with the latter corresponding to a negative scenario as previously explained in chapter 2, Section 2.3.7.

In the next section, we present the UML viewpoint of the structures in Figure 5.4.

5.2.2 The UML Viewpoint

From the UML point of view, our profile specializes primarily the SPT Profile, which defines the notion of time and resources modeling. In this Section, we combine the reliability domains in Sections 5.2.1.1 and 5.2.1.2 into one consistent UML viewpoint.

In Figure 5.5 we show how the elements of our analysis domain model relate to the elements that constitute the SPT Profile. The elements in italics with grey shading are part of the SPT Profile and each of them can find a corresponding equivalence with those in Figure 5.3, except for *Stimulus*. It was not included in Figure 5.3 for the sake of simplicity, but it is still con-

sistent with the Dynamic Usage Model of the SPT. Here we also explicitly model the Service Delivery as an specialization of the QoS value and the RelActiveResource as a ResourceInstance sub-type. Note we do not mention in Figure 5.5 the RelPassiveResource since we are modelling the dynamic perspective of the reliability analysis context.

Note that all the elements in our domain model, except for *Node* and *Scenario*, extend elements of the SPT Profile. Both of the abstract elements *Node* and *Scenario* of our domain, depicted in Figure 5.4, can be represented as a *Scenario* in the SPT profile.

A *Scenario* (from the SPT) comprises ordered series of steps (or ActionExecutions) and is specialized in our reliability analysis context as an *HMSC*, a *BMSC* and a *NegBMSC*. A *BConnector* of our domain extends both functionalities of *ActionExecutions* and *Stimulus*. ActionExecutions as a level of abstraction of Scenarios (from the SPT) is the super-type of our BConnector, which may encapsulate scenarios through a single message exchange between component resources.

As a Stimulus, the BConnector represents an instance of communication in transit between a calling object and a called object (cause-effect relationship). In the relationship with the QoS value, a BConnector specifies the required QoS value (as service delivery) that the communication channel of the message exchanged needs in order to meet its own obligation.

An *HTransition* is also an extension of Stimulus as a scenario transition can be triggered in the occurrence of an event for a certain ActionExecution (the *generates* association in Figure 5.5). The *BComponent* specializes the *RelActiveResource* (a sub-type of *ResourceInstance*)

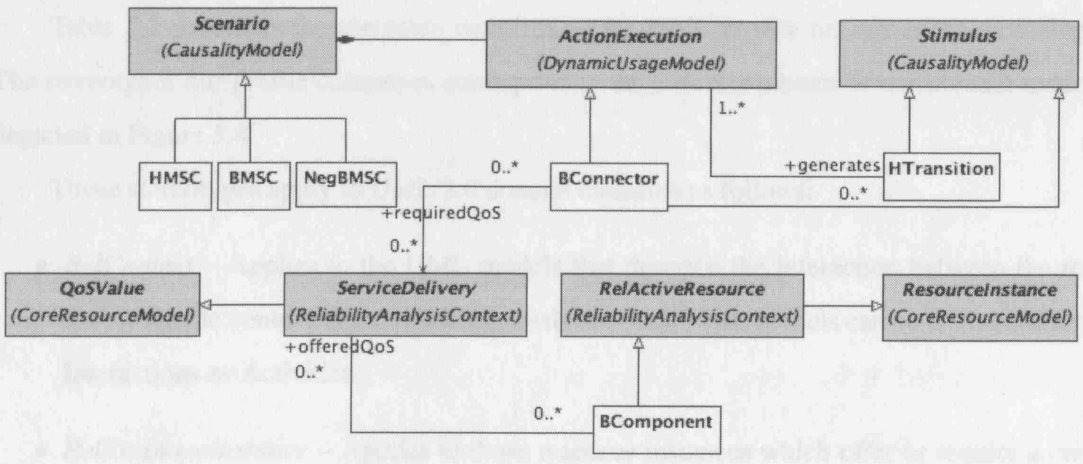


Figure 5.5: Relationship between Our Reliability Analysis Profile and the SPT Profile.

and as such, it is expected to offer a QoS value as service delivery (*offeredQoS* association in

Figure 5.5).

Stereotype	*Base Class*	*Tags*
≪ *RelContext* ≫	Interaction Activities Collaboration	–
≪ *RelResourceInstance* ≫	Instance Component Node	RequiredValue OfferedValue RelMetric
≪ *HMSC* ≫	Interaction Activities	HName
≪ *BMSC* ≫	Interaction	BName
≪ *NegBMSC* ≫	Interaction	BName
≪ *HTransition* ≫	Interaction	PTS
≪ *BComponent* ≫	Classifier Component Instance	BCompRel
≪ *BConnector* ≫	Stimulus Message Connector *(from InternalStructures)* Connector *(from BasicComponents)*	BConnRel
≪ *Stop* ≫	Interaction	–

Tag	*Type*	*Multiplicity*
RelMetric	Enumeration{ 'ROCOF', 'POFOD', 'MTTF', 'AVAIL'}	[1]
RequiredValue	Real (0,1]	[0..1]
OfferedValue	Real [0,1]	[0..1]
PTS	Real (0,1]	[0..1]
BCompRel	Real (0,1)	[0..1]
BConnRel	Real (0,1)	[0..*]
HName	String	[0..1]
BName	String	[0..1]

Table 5.1: Stereotypes and Tag Definitions for the Reliability Analysis Profile.

Table 5.1 describes the elements constituting the UML profile for reliability modeling. The stereotypes our profile comprises correspond to the concrete classes of our domain model depicted in Figure 5.4.

Those stereotypes apply to UML 2.0 domain elements as follows:

- *RelContext* – Applies to the UML models that describe the interaction between the resources in the context of the reliability analysis. Such UML models can be Collaboration, Interactions or Activities.

- *RelResourceInstance* – Applies to those resource instances which offer or require a certain level of reliability in the reliability context. As active resources, which are involved in the dynamics of the reliability analysis, this stereotype is applied to UML elements involved in describing the dynamics of the system such as Instances, Components and the Nodes where they are deployed. Their offered or required QoS value are also associated to a certain reliability metric.

- *BMSC* – Applies to *Interactions* of *Sequence Diagram* type.
- *NegBMSC* Applies to *Sequence Diagrams* with a *CombinedFragment* having *neg* as its *InteractionOperator*.
- *HMSC* – Applies to the *Interaction Overview Diagram*, which is the structure that best suits the modeling of an HMSC. *Interaction Overview Diagrams* focus on the overview of the flow of control where the nodes are *Interactions* or *InteractionOccurrences* [OMG04a]. Also, as a structure to represent the flow of control, the *Interaction Overview Diagram* enables the representation of the initial and the final states of the flow, which are also structures required in our reliability prediction technique [RRU05c] (as described in Chapter 4). Alternatively, we could use the *CombinedFragments* structure, but an *Interaction Overview Diagram* is semantically closer to HMSCs.
- *HTransition* – Applies to an *Interaction* representing an alternative choice of behaviour. It is tagged with the value *PTS*, the probability of transition to the *Interaction*.
- *BComponent* – Applies to components participating in *Sequence Diagrams* to be analysed by the model processor. The tag *BCompRel* associated to the *BComponent* stereotype represents the reliability of the component in our reliability prediction technique.
- *BConnector* – Applies to messages exchanged between two *BComponents* in an *Interaction*. The tag *BConnRel* associated to the *BConnector* represents the reliability of the connector enabling the communication between the components. The reliability of the connector is regarded as the probability of success of a message transition, irrespective of the transition execution time. Notice that from Table 5.1 it can be applied to two UML *Connector* types: one that extends from *InternalStructures* and one that extends from *BasicConcepts*. The reason is that *Connector* from InternalStructures specifies a link that enables communication between two or more *instances*, where the link may be as simple as a pointer or as complex as a network connection. While as a *Connector* from *BasicConcepts*, it can either link the external contract of a *component* to that component's realization or connect two *components* where one provides a service the other component require (e.g., as a client-server connection).
- *Stop* – Due to the assumption in our prediction technique that there must be one final scenario in the scenario specification [RRU05c] (as described in Chapter 4), it is required that no more than one *Interaction* connects to the *final node* of the HMSC *Interaction Overview Diagram*. The *Stop* stereotype applies to the *Interaction* with that feature.

The following are constraints defined in our reliability analysis profile package:

Rule 1 Within an *Interaction Overview Diagram* stereotyped as an *HMSC*, every node must be either a *BMSC* or another *HMSC*.

Rule 2 Every *HMSC* and *BMSC* must be uniquely named.

Rule 3 Every *HMSC* must have one *Activity initial node* and one *Activity final node* (stereotyped as *Stop*) only.

Rule 4 *HMSC* nodes must have at least one incoming and one outgoing transition, except the *initial node* and *final node*.

Rule 5 The *PTS* values of *HTransition*-stereotyped nodes connected to the same *Decision* node within an *HMSC* must sum to one.

5.2.3 Mapping From UML to LTSA

Once our profile is applied to a UML model, the translation from UML to LTSA is carried out. The transformation consists of (1) parsing the XML Metadata Interchange (XMI) form of the UML model, which is the standard representation of UML models in XML [OMG02b], and (2) generating the XML input format accepted by LTSA.

Current UML tools provide only partial conformance with the UML 2.0 specification, which has forced us to make some workarounds in our implementation. The major problem we encountered was to apply the stereotype *HTransition* and its *PTS* tagged value to the nodes (i.e., *Interaction Occurrences*) within *Combined Fragments* within *Interaction Overview Diagrams*. To get around this problem, we had to associate the *HTransition* stereotype with the transitions between nodes rather than to the nodes themselves. This solution is temporary, and we will evolve the implementation of our profile as tool support improves to properly accommodate the UML 2.0 specification.

We implemented the transformation of our UML profile to LTSA in XSLT [W3C99]. XSLT describes rules for transforming a source document in a tree format (such as an XML file) into a result document described also by a tree. It therefore suits our need to transform the XMI representation of a UML model into the XML format accepted by LTSA. The transformation process is rather straightforward as long as the following conditions are satisfied:

1. An HMSC in LTSA cannot have multiple nodes that correspond to the same BMSC. In case there are multiple *Interaction Occurrences* of the same *Sequence Diagram* in a UML *Interaction Overview Diagram*, those multiple occurrences are reduced to just one node

of the LTSA HMSC during the transformation process, keeping the same set of transitions contained in the *Interaction Overview Diagram.*

2. LTSA does not support hierarchically nested HMSCs at the moment. In case an *Interaction Overview Diagram* is specified in multiple hierarchical levels, it should be flattened before transformation is carried out.

5.2.4 Mapping Analysis Results Back into UML

After analysis has been carried out in LTSA following the approach presented in Chapter 4, we have the system reliability prediction and the detection of implied scenarios. In particular, we can use this analysis to provide answers to the following questions: Do we have any implied scenarios in our system architecture model? What is the impact of the implied scenarios on the system reliability? What is the sensitivity of the system reliability to changes in individual probability values?

If an implied scenario is a positive scenario, which means that the detected trace is to be included in the scenario specification, then a new Sequence Diagram is constructed for the trace and annotated with our profile for reliability prediction. This new *Interaction* is then incorporated appropriately as a node in the *Interaction Overview Diagram.* Incoming and outgoing transitions must be manually attached to the new positive scenario. If an implied scenario is a negative scenario, i.e., a trace to be avoided, it needs to be incorporated into a *NegBMSC*, with the undesirable message traces specified inside an *Interaction Fragment* having *InteractionOperator* type *neg*.

The analysis reveals how the system reliability is sensitive to (1) the *component reliabilities*, and (2) the *scenario transition probabilities*, as we presented in previous chapter. These two analyses can help in identifying components and scenario transitions that could threaten the reliability of the software system. The results produced by the sensitivity analysis can then be used by system designers to decide on mechanisms to use for enhancing the system reliability.

5.2.5 Example

We exemplify the application of our approach using a variant of the Boiler Control system introduced in Chapter 3. The difference now is that, in this section, we specify our Boiler example using our Reliability Analysis package.

The Interaction Overview Diagram of Figure 5.6 shows that the Boiler Control system composes five Sequence Diagrams *Initialise*, *Register*, *Analyse*, *Terminate* and *Shutdown*, which are are depicted in Figure 5.7.

As presented in Section 5.2.2, the stereotype *HTransition* is tagged with the probability of

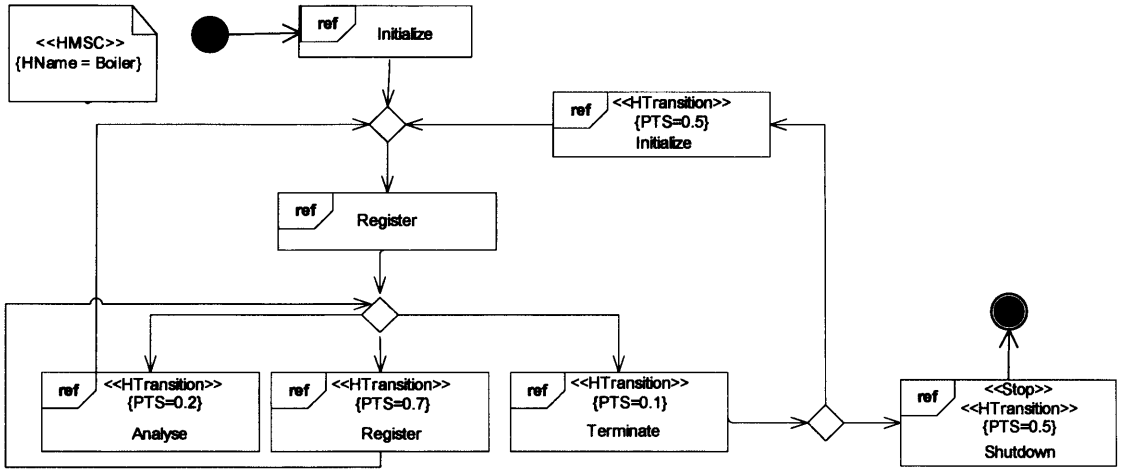


Figure 5.6: The Interaction Overview Diagram of the Boiler System.

transition between scenarios, *PTS*, as shown in Figure 5.6. The values for the *PTS* are based on the assumption that the system executes the scenario *Register* (which causes sensor readings to be entered into the database) far more frequently than the scenarios *Analyse* and *Terminate*, and that when it does execute *Terminate* there is an equal probability of either reinitialising or shutting down completely. As shown in the figure, it may be necessary to specify multiple references to the same Sequence Diagram if they are to be tagged with different scenario transition probabilities.

Inside the *BMSC*-stereotyped Sequence Diagrams, the components' reliabilities are annotated by applying the stereotype *BComponent* with its tagged value *BCompRel*, as depicted in Figure 5.7. Without loss of generality, we use coarse-grained, single values for the overall component reliabilities. In general, we can also associate finer-grained values for reliability through annotation of individual messages and segments of component timelines. The *BConnector* element of our profile suits the use of finer-grained values where individual messages can also be associated with a communication reliability value; in the example, these values are all set to 1.0. Notice that the *Shutdown* Sequence Diagram is not present in Figure 5.7, as it has traces identical to those in the *Terminate* scenario.

Following the steps of our reliability prediction technique, as presented in Chapter 4, the LTS model for each component participating in the scenarios is generated;Then, the *Architecture Model* of the system is synthesised as the parallel composition of the component LTSs, followed by the computation of a prediction for the system reliability for the synthesised Architecture Model.

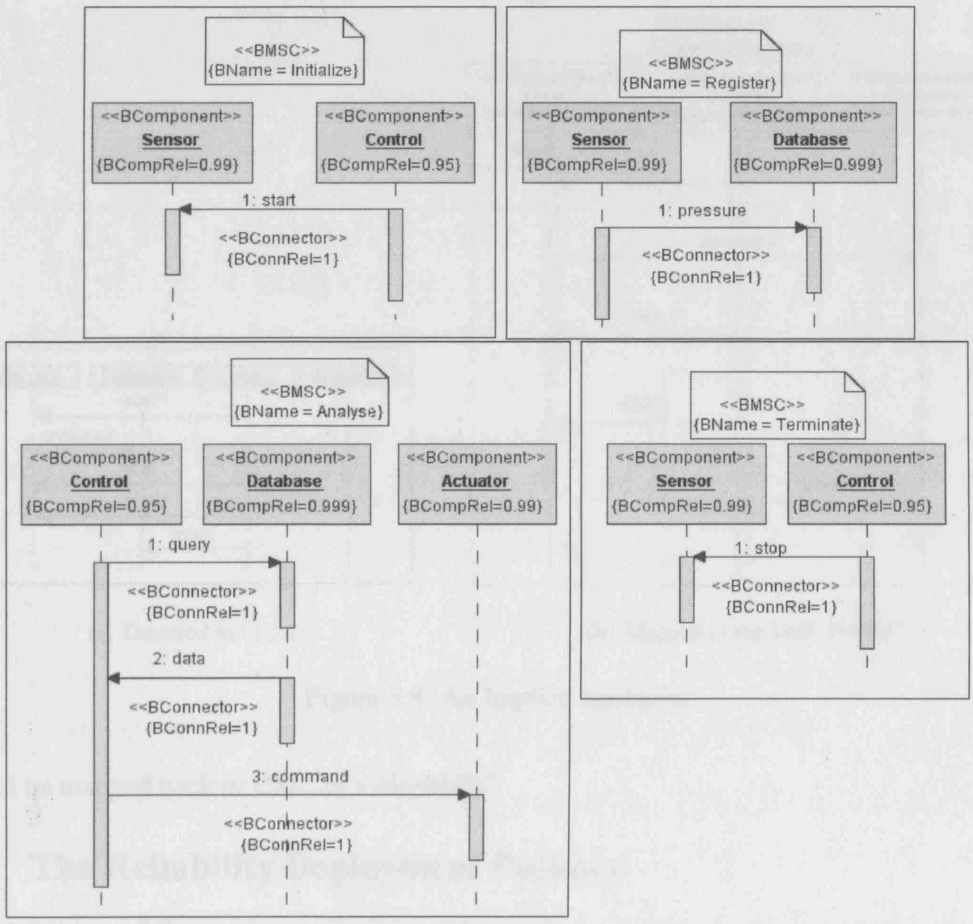


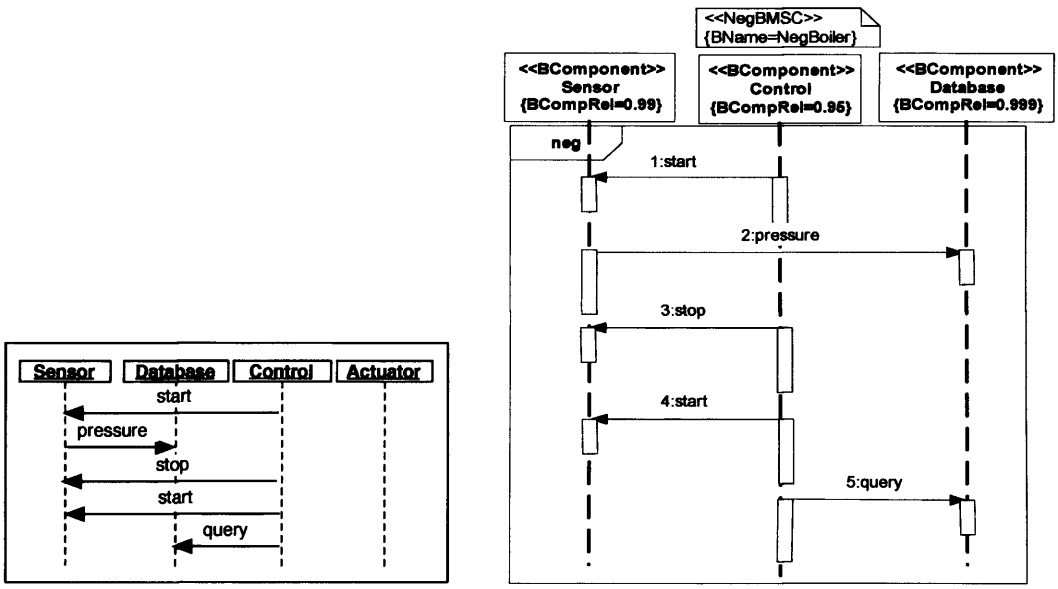
Figure 5.7: The Annotated Sequence Diagrams of the Boiler System.

5.2.5.1 Validating for Implied Scenarios

As also presented in chapter 4, the Boiler Control System specification of Figures 5.6 and 5.7, has implied scenarios, and Figure 5.8(a) depicts one of them. From the specification, we found out that the Boiler Control system architecture exhibits the trace *start–pressure–query–data–command*, and that component *Control* interacts with *Database* only through messages *query* and *data*.

In other words, the Architecture Model produces a trace that reveals a mismatch between behaviour and architecture, and we view this particular trace as being undesirable. We follow the same approach presented on chapter 4, where this trace represents a negative scenario, and so a set of constraints preventing the occurrence of the negative scenario is expressed in FSP, the modeling notation of LTSA [JK99], and then composed with the Architecture Model.

Following the steps of our reliability prediction technique, a *Constrained Model* of the system is then synthesised as the parallel composition of the constraints with the Architecture model previously obtained. Figure 5.8(b) depicts the implied scenario detected in LTSA as it



(a) Detected in LTSA (b) Mapped to the UML Profile

Figure 5.8: An Implied Scenario.

would be mapped back to UML as a *NegBMSC*.

5.3 The Reliability Deployment Package

The definition of the deployment profile follows the realization mapping between client and supplier, as we show in Figure 5.9. From the Figure, it can be noticed that the realization mapping allows one or more supplier elements to be realized by one or more client elements. In order to better capture the mapping between clients and suppliers, the realization mapping in the GRM is specialized into three other sub-types: (1) *code* - to denote the client physically contains the program code for the supplier; (2) *deploys* - to denote that instances of the supplier are located on the client; (3) *requires* - a specialization of the *deploys* mapping, used to indicate that if the actual deployment environment cannot satisfy the minimum required QoS, it is not possible to assure that the supplier will provide its offered QoS.

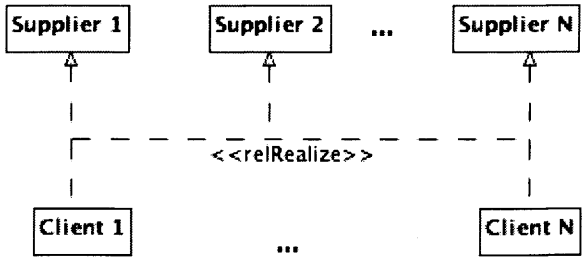


Figure 5.9: Client-Supplier Semantics.

We define a deployment package consistent with the realization mapping as follows:

1. `relCode` - A mapping that extends the *code* realization from the *RealizationModelPackage* to denote the client physically contains the code for the supplier, where the supplier is part of the model element and the client is either a UML component or a node.

2. `relDeploys` - A mapping that extends the *deploys* realization from the *RealizationModelPackage* which indicates that the supplier's instance is located on the client. Equivalently, component *CompX* in Figure 5.10 could have been placed inside *NodeX* to indicate the component is deployed on that node. Therefore, this stereotype requires two tag definitions: `Node`, which assigns a name for the each node of the cluster and `Address`, which assigns the host address of the node in the network.

3. `relDeploymentSpec` – Similar to the *deployment spec* from the *Deployments* package from UML 2, which is an Association class of the stereotyped `relDeploys` association, containing the parameters of the deployment descriptor files.

4. `relRequires` - Extends the *requires* realization from the *RealizationModelPackage* and indicates the client represents a generic specification of the minimal acceptable deployment environment required by the supplier. This mapping may be also applied to abstract the complex mechanisms of fault tolerance to assure the desired level of software availability from the supplier is achieved in the execution environment (the client).

5. `relConnector` - This is an element to represent the reliability (as QoS value) of the connector (a passive resource instance) that enables the communication between the components. This stereotype can be applied to two UML *Connector* types: one that extends from *InternalStructures* and one that extends from *BasicConcepts*. The reason is that *Connector* from *InternalStructures* specifies a link that enables communication between two or more *instances*, where the link may be as simple as a pointer or as complex as a network connection. Being a *Connector*from *BasicConcepts*, it can either link the external contract of a *component* to that component's realization or connect two *components* where one provides a service that the other component requires (e.g., as a client-server connection). It has `relConnValue` as a tag definition which stores the reliability value of this connector.

An example of the application of most of the profile elements is depicted in Figure 5.10, which is divided into two blocks: the supplier model (upper block), which contains the model elements of the required set of QoS values; and the client model (the lower block), which

contains the configuration information of the deployment context and its offered QoS values. Note that we annotate classes A and B with a new stereotype $\ll$ *Resource* $\gg$ just to model those as *static* elements, in accordance with the Core Resource Model concepts.

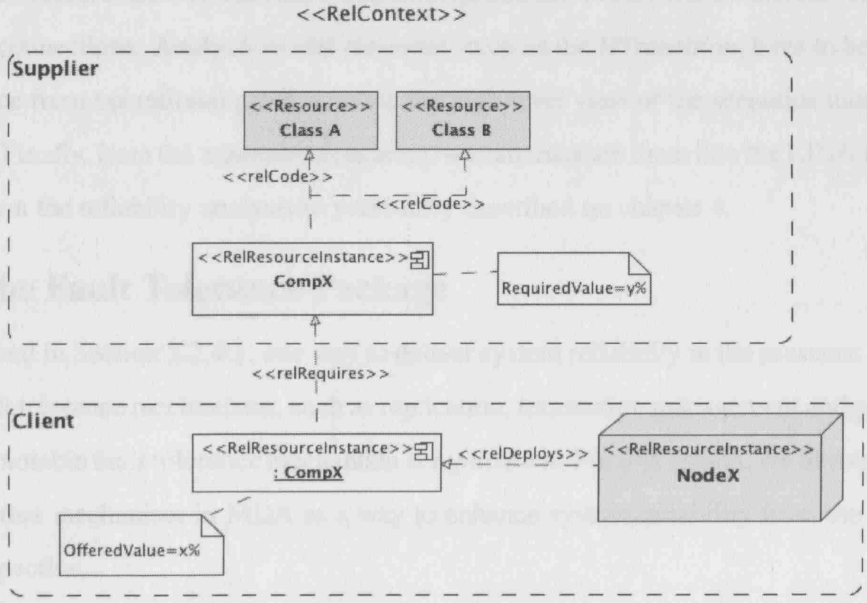


Figure 5.10: An Abstract Example of Deployment Profile Elements.

5.3.1 Mapping Analysis, Design and Deployment

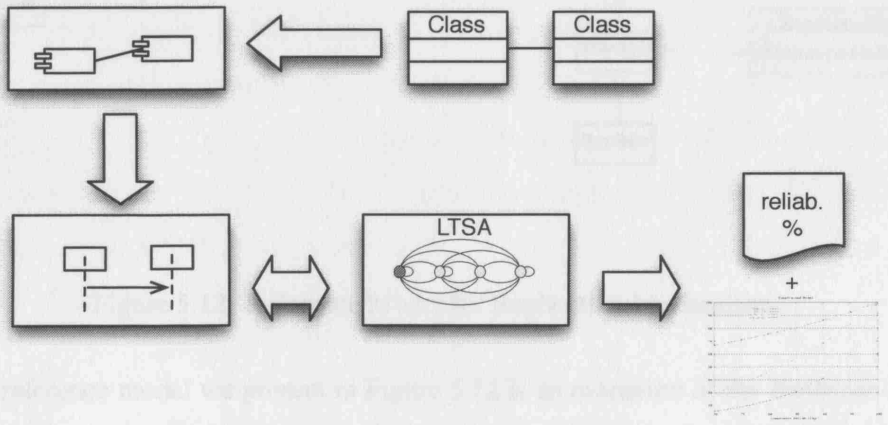


Figure 5.11: The Mapping Overview.

In order to make the analysis, design and deployment integrated into a seamless environment, we make use of common UML environment shared by the analysis and the design profiles. Most of the information required for the analysis model can be directly obtained from the information annotated on the component model that both analysis and design have in com-

mon, as presented in Figure 5.11. From the software specification, classes are mapped into components and annotated with the profile for reliability, with stereotypes and tag definitions mapped as appropriate. From components view, the set of scenarios describing their interactions can be constructed and annotated with information derived from the annotated components and their connections. Analysis model elements, such as the *HTransition*, have to be provided, for instance from operational profiles, when the high-level view of the scenarios interactions is modeled. Finally, from the annotated scenarios, we can translate them into the LTSA framework and perform the reliability analysis as previously described on chapter 4.

5.4 The Fault Tolerance Package

As described in Section 2.2.4.1, one way to deliver system reliability in the presence of faults is to use fault tolerance mechanisms, such as replication, transaction management and persistency. The most notable fault tolerance mechanism is *replication*. For this reason, we devote our focus to model that mechanism in MDA as a way to enhance system reliability from the modelling level perspective.

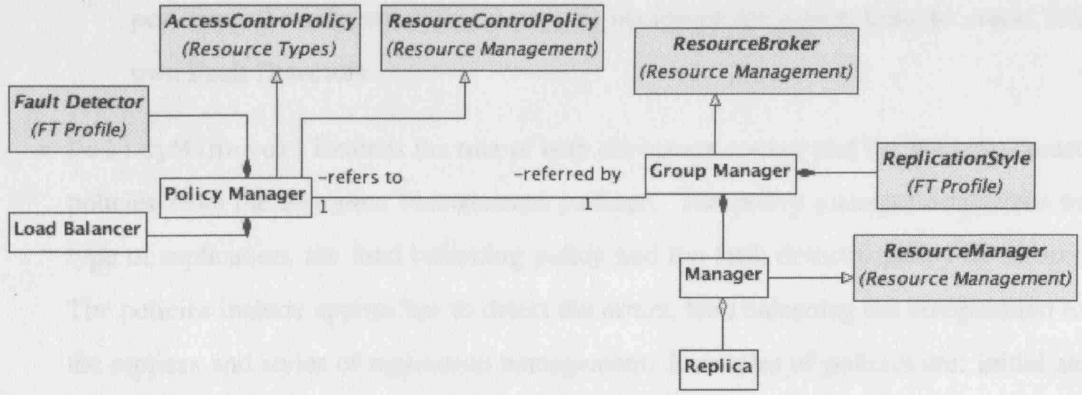


Figure 5.12: Reference Model for Replication Mechanisms.

The reference model we present in Figure 5.12 is an extension of the Resource Management package of the GRM. It also contains elements that are common to the Generic Fault-Tolerant framework in [OMG04b] and constitutes one of the possibilities to achieve fault tolerance in the context of component-based middleware. In the platform specific level, that framework holds a structure similar to the CORBA FT architecture due to the latter's comprehensiveness. However, the model we present in this section is not only applicable to fault-tolerant CORBA, but also to the EJB model, as we shall see in Section 5.4.3.

The profile for replication comprises the following elements:

- `Load Balancer`: Specify policies to determine the server node which is going to receive new incoming requests. A priori, we define only two policies: `round-robin` and `first available`. A round-robin policy dispatches invocations to randomly selected nodes. The first available policy selects one available target node and uses it for every invocation.

- `Fault Detector`: Specify policies which describe the mechanisms safety engineering uses to describe how to monitor software faults. It can be of three different types:

  - `StaticallyDeployedFaultDetectors`: In an operating environment with a relatively static configuration,location-specific Fault Detectors will typically be created when the FT infrastructure is installed.

  - `InfrastructureCreatedFaultDetectors`: The FT infrastructure may create instances of the Fault Detectors to meet the needs of the applications. Therefore, the infrastructure is configured to match those needs.

  - `ApplicationCreatedFaultDetectors`: For some reason (e.g lack of support from the infrastructure), it may be necessary for applications to create their own Fault Detectors.

- `PolicyManager`: Extends the role of both the access control and the resource control policies from the Resource Management package. The policy manager establishes the type of replication, the load balancing policy and the fault detector policy to be used. The policies include approaches to detect the errors, load balancing the computation for the replicas and styles of replication management. Examples of policies are: initial and minimum number of replicas (i.e. creation of resource instances), 'first-available' or 'round-robin' for the load balancing (i.e. allocate/deallocate resources) and styles of replication management (i.e., transient or persistent state).

- `GroupManager`: Extends the role of the resource broker from the Resource Management and has an equivalent element in the QoS profile [OMG04b] named *ServerObjectGroup*. It manages the object group of a fault-tolerant object by managing the several `Manager` instances. A reference to the entire group makes transparent to the clients the concept of replication. Contains a reference to replica states for the synchronization between primary and backups (for replication style), and request order (for load balancing).

- `Manager`: Extends the role of the resource manager. It is used by the `GroupManager` to detect faults and to create a new instance of a replica as a Factory object, where

each replica has its own identity. Each group's `GroupManager` has one or many `Manager` instances: one instance for each partition where replicas are deployed. The set of `Manager` instances is controlled by a `GroupManager`. In the case of a fault detected by the `Manager`, the `GroupManager` automatically reconfigures the system by reassigning task among non-failed components. Then the `GroupManager` will try to reconstruct that failed component by using the factoring mechanism to recover the components to a saved state prior to error detection.

- `Replica`: Extends the features of a *RelResourceInstance*, which is created, managed and (de)allocated by the resource broker and the resource manager. It also has an equivalent element in the QoS profile [OMG04b] named *ObjectReplica*. Conceptually, the replica is a redundant entity and contains attributes such as state (as primary, backup or transient), style (as transient or persistent), status (allocated or not) and location. Additionally, it contains the following tag definitions: `ReplicaState`, `ReplicationStyle`, `Clustered` (to indicating if it is already allocated in a cluster), `ReplicaID` and `ReplicaLocation`.

- `ReplicaState`: Defines the dynamic state information of a Replica, which depends on the role of the replica in the group. This role determines whether the replica is *primary, backup* or *transient* and it will vary according to the policy used.

- `ReplicationStyle`: A replica can be made redundant through two different styles: *transient* or *persistent*. For this reasion, this metaclass is specialized in two other elements:

  - `TransientStateReplicationStyle`: This replication style defines styles for objects having no persistent state. For `StatelessReplicationStyle`, the one kind of `TransientStateReplicationStyle`, the history of invocation does not affect the dynamic behavior of the object group.

  - `PersistentStateReplicationStyle`: This style defines how objects that have a persistent state will be replicated. Two distinctive persistent replication styles can be identified:

    1. `PassiveReplicationStyle`: Requires that only one member of the object group, the primary member, executes the methods invoked on the group. Periodically, the state of the primary member is recorded in a log, together with a sequence of method invocations. In the presence of a fault, a backup

member becomes a primary member. The new primary member reloads its state from the log, followed by reapplying request messages recorded in the log. When it is cheaper transferring a state other than executing a method invocation in the presence of a failure and the time for recovery after a fault is not constrained, passive replication is worth considering. The style of the `PassiveReplicationStyle` can be `Warm` or `Cold`. In the `Warm` style, the state of the primary member is transferred to the backups periodically. While in the `Cold` style, the state of the primary member is loaded into the backup only when needed for recovery.

2. `ActiveRepicationStyle`: A second type of persistent replication which requires that all the members of an object group execute each invocation independently but synchronously. When it is cheaper to execute a method invocation rather than transferring a state or when the time available for recovery after a fault is tightly constrained, active replication is worth considering. If the voting attribute is active, the requests from the members of the client object group are voted, and are delivered to the members of the server object group only if the majority of the requests are identical. The situation is analogous to replies from the members of the server object group.

Based on this reference model, we are able to complete the profile with stereotypes and tag definitions, as shown in Table 5.2. The profile elements for fault tolerance are organized in Table 5.2.

5.4.1 Mapping the Replicas to the Deployment Profile

In this section we define rules to model fault tolerance in the deployment level of component-based systems. These rules apply to the deployment profile we defined in Section 5.3, where the core element `Replica` derives from the `RelResourceInstance`. It is worth noting those rules are expressed in OCL in Appendix A.

Thus, the rules are the following:

Rule 1 – For a replica to be clustered, the `Replica` stereotype must have its `Clustered` tag `True`.

Rule 2 – For each deployed replica of the same component, the node name and the host address should be unique.

Rule 3 A `relDeploys` must have as client and supplier a `Replica` base class.

Stereotypes	*Base Class*	*Tags*
≪ *GroupManager* ≫	Class Component Instance Node	ReplicaState ReplicationStyle
≪ *PolicyManager* ≫	Classifier Component Instance Node	FaultDetectorPolicy LoadBalancePolicy
≪ *Manager* ≫	Classifier Component Instance Node	ManagerID ReplicationStyle ReplicaState
≪ *Replica* ≫	*RelResourceInstance*	ReplicaState ReplicationStyle ReplicaLocation ReplicaID Clustered

Tag Name	*Tag Type*	*Multiplicity*

FaultDetectorPolicy	Enumeration{
	'StaticallyDeployed'
	'InfrastructureCreated'
	'ApplicationCreated'
	}
		[0..1]

LoadBalancePolicy	Enumeration{ 'FirstAvailable 'RoundRobin }	[0..1]
ManagerID	String	[0..1]
ReplicaState	Enumeration{ 'Primary' 'Backup' 'Transient'}	[0..1]
ReplicationStyle	Enumeration{ 'TransientStyle' 'PersistentStyle'}	[0..1]
TransientStyle	Enumeration{ 'Stateless' 'Stateful'}	[0..1]
PersistentStyle	Enumeration{ 'PassiveReplication' 'ActiveReplication'}	[0..1]
ReplicaLocation	String	[0..1]
ReplicaID	String	[0..1]
Clustered	Boolean	

Table 5.2: Stereotypes and Tag Definitions of the Profile for Fault Tolerance.

Rule 4 – The offered and the required reliability assurance of each component must be defined according to what is provided in client and supplier of a `relRequires` realization mapping. By this means, it is possible to quantify the number of replicas each components should have in the cluster to guarantee their required reliability level. Due to the reasonable complexity to define this rule in particular, we provide in the next sub-section more details about it.

5.4.2 Quantifying the Replicas in the Deployment

As the environment of the required and offered quality of service is set, one can now find out how many components are needed so that the reliability offered by the environment can meet the reliability required on the system specification. For this purpose, we use the approach where components are independently grouped, in a parallel configuration. To improve components' reliability through the underlying structure at the deployment level we apply independent components replication configuration as described in [MIO87, MA01]. Following that approach,

the reliability of the parallel system, R_p, is the probability that at least one component is in operation when needed. So, the probability that all n components are down is then:

$$Pr[\text{all components are down}] = \prod_{i=1}^{n}(1 - r_i) \tag{5.1}$$

where r_i is the reliability of each component. So, assuming independence of failures between components, we obtain:

$$R_p = 1 - Pr[\text{all components are down}]$$
$$R_p = 1 - [(1 - r_1) \times (1 - r_2) \times \ldots (1 - r_n))] \tag{5.2}$$

When all the component replicas have the same reliability, the functional formula for this assurance is:

$$R_p = 1 - (1 - r)^n > a \tag{5.3}$$

where r is the reliability of each component, a is the required reliability of the system and n is the number of components that should be composed in each cluster.

To reflect this scenario, the classes of the design profile to be replicated should be mapped to the deployment profile through the mapping profile specified in OCL in Appendix A, Section A.2. As we assume the reliability is provided by the underlying middleware, the offered reliability for the component is specified on the deployment diagram, while the required reliability comes from the value specified elsewhere the class model, for instance in the component's model. The realization mapping from the Deployment model from Section 5.3 that expresses this requirement is the $\ll$ *relRequires* $\gg$.

5.4.3 The Replication PSM for EJB

In this section we relate the profile for replication to the EJB context. Prior to defining the PSM for replication in EJB, we identify the elements to realize a profile for replication in EJB by configuring and deploying an application using JBoss which provides a wide range of clustering features for stateless session beans, stateful session beans, entity beans and JNDI [LB04]. The sample application we used for that purpose is the Duke's Bank online banking application [BGJP05] in Section 5.4.4. Once the elements for replication are identified we move towards the definition of the EJB profile for replication in Section 5.4.5.

5.4.4 Clustering the Duke's Bank

The Duke's Bank is an online bank application implemented for the J2EE tutorial. It has an administrator client to manage customers and accounts and a web client where customers access account histories and perform transactions. We depict the high-level view of the components

interaction in Duke's Bank in Figure 5.13. The clients personal information and transactions are maintained in a database through enterprise beans. The J2EE server used in our example was JBoss, as previously mentioned, which then uses Tomcat as its web container. The database used was MySQL.

The Dukes Bank application has three session beans:

- AccountControllerBean
- CustomerControllerBean
- TxControllerBean[1]

These session beans provide a clients view of the applications business logic. Hidden from the clients are the serverside routines that implement the business logic, access databases, manage relationships, and perform error checking.

For each business entity represented in the Dukes Bank application, there is a matching entity bean:

- AccountBean
- CustomerBean
- TxBean

Each bean provides an object view of the database tables: *account*, *customer*, and *tx*. For each column in a table, the corresponding entitybean has an instance variable. Because they use container-managed persistence(CMP), the entity beans contain no SQL statements that access the tables. The enterprise bean container manages all data in the underlying data source, including adding, updating, and deleting data from the database tables.

The central concept for clustering in JBoss is the *partition*, which is the way several JBoss server instances are grouped. JBoss does not require to statically define a cluster topology. By setting up the cluster name, i.e., the partition name, any node configured for the same partition can dynamically join or leave the partition, with some delay though, according to the multicast configuration and network load. From Figure 5.14, it can be noticed the JBoss clustering architecture is divided in three major layers [LB04]:

1. Layer 1: JavaGroups for the multicast communication framework. As JavaGroups, or in shortly JGroups, is concerned with the reliable multicast protocol and not the middleware

[1]Tx stands for a business transaction, such as transferring funds

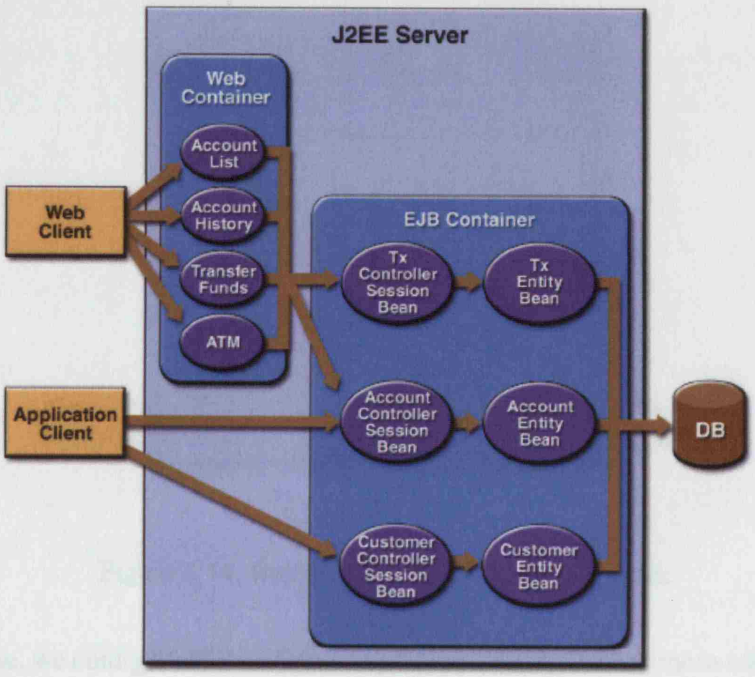


Figure 5.13: The Duke's Bank Architecture Overview [BGJP05].

itself, we do not go into much detail about it here. More information about JGroups can be obtained at [B. 06].

2. Layer 2 - HAPartition(HAP) to abstract the communication framework and provide access to some basic communication primitives. The HAPartition contains (1) the Distributed Replicant Manager (DRM) to manage the list of serialized and replicated data (e.g., RMI stubs) owned by a specific node, (2) the HASessionState - a cluster-wide distributed service to manage Stateful Session Bean state, and (3) the Distributed State Service (DS) to allow sharing a set of dictionary throughout the cluster

3. Layer 3 - High Availability (HA) frameworks that abstract away the complexity to configure server objects(e.g., RMI, EJB, JMS) and the java naming server (JNDI).

Therefore, in order to configure a cluster in JBoss, one needs to verify the attributes to specify for each layer in Figure 5.14. In Figure 5.15 we present the major attributes required to configure the cluster and its nodes. For the Duke's Bank application, we configured the minimum set required, where default values were maintained as provided through the JBoss distribution. In this case, the attribute HAPartition (HAP) name was kept as *DefaultPartition* and attribute JavaGroups multicast address 228.1.2.3. Both attributes are configured in cluster-service.xml. As we wanted to use Tomcat features as web server, we also had to configure the *server.xml* file attribute *jvmRoute*, which names the nodes of a cluster, i.e., node1 and node2. As

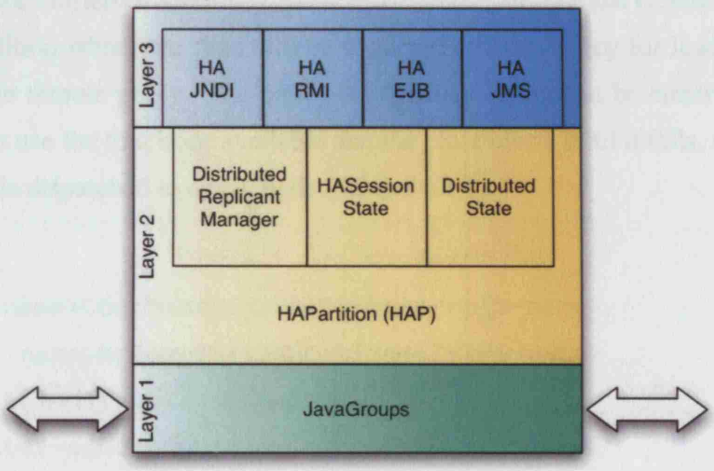


Figure 5.14: Building Blocks for a JBoss Cluster.

for the database, we configured JBoss to use MySQL database as it has more robust features for database management in terms of data synchronization and transaction management compared to the Hypersonic one that comes along with JBoss distribution.

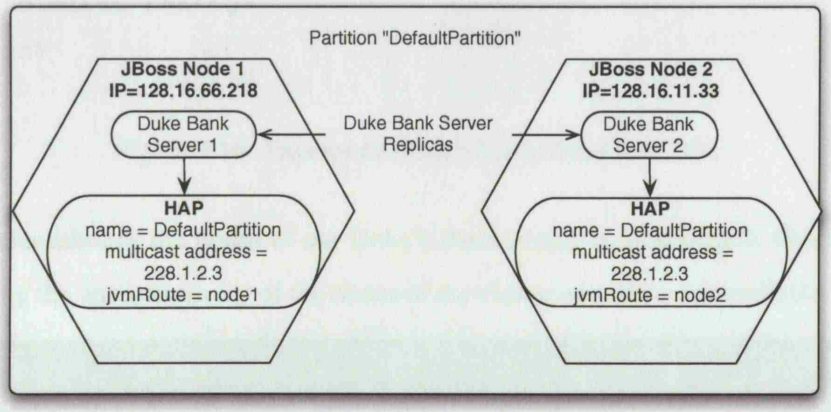


Figure 5.15: Duke's Bank Cluster Configuration Overview.

The major advantage of using a J2EE server compared to a CORBA server is the abstraction of configuration complexity. On a J2EE server like JBoss, in order to configure the application to make it fault-tolerant, there is almost no need to change the original code. For the Duke's Bank application for instance, the only modification on the code required was to make serializable those classes that did not extended EJB classes. This was because some classes in the Tomcat web server container needed to be serialized so that they could communicate with the other replicas throughout the cluster. The major configuration for the Duke's Bank session and entity beans takes place in just one file: jboss.xml (see Figure 5.16. For each bean to be replicated, it is required to have the tag `<clustered>`, which is mandatory to indicate that

the bean will work clustered. The <cluster-config>, an optional element, configures the name of the partition where the bean will be clustered and the policy for load-balance for the home and for the remote proxy. The policy for load balancing can be either *First Available*, which persists to use the first node available that the class meets until it fails, or *Round-Robin*, where each call is dispatched to a new node (the default).

```
<session>
      <ejb-name>CustomerControllerBean</ejb-name>
      <jndi-name>MyCustomerController</jndi-name>
      <clustered>True</clustered>
      <cluster-config>
            <partition-name>DefaultPartition</partition-name>
            <home-load-balance-policy>
                  org.jboss.ha.framework.interfaces.RoundRobin
            </home-load-balance-policy>
            <bean-load-balance-policy>
                  org.jboss.ha.framework.interfaces.RoundRobin
            </bean-load-balance-policy>
      </cluster-config>
</session>
```

Figure 5.16: Excerpt of Duke's Bank Jboss.xml File.

The screenshot of the nodes of our Duke's Bank cluster is presented in Figures 5.17 and 5.18 showing the auto-discovery of the nodes of the cluster once they are available.

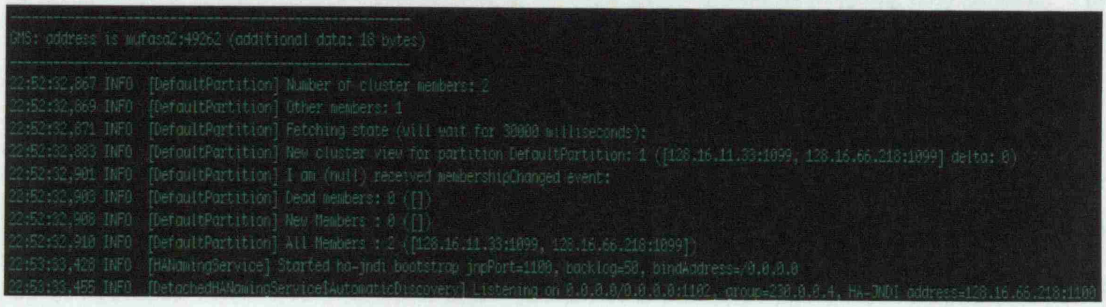


Figure 5.17: Screenshot of the Auto-Discovery in Node 1: mufasa2 (128.16.66.218).

There were two problems we noticed in our experiment though. They are both related to the hot deployment feature of JBoss. The first problem is that once a node left the cluster, and wants to join the cluster again, it does not occur on a deterministic basis. Sometimes, the

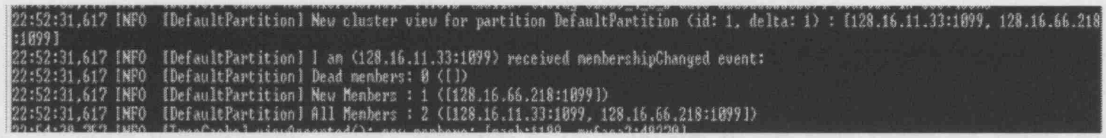


Figure 5.18: Screenshot of the Auto-Discovery in Node 2: mash (128.16.11.33).

cluster recognizes as a new joining node and accepts that re-joining node. Other times it does not happen. We noticed this problem could be somewhat turned around when we cleaned the *temp* folder under the deployment directory at node mash, which was running under Windows 2000 operating system, while mufasa2 was running under MacOS X. The other problem we realized is already mentioned in the JBoss clustering document [LB04], where it is not possible to hot-deploy services based on HAPartition. but it is possible to hot-deploy *new* HAPartitions and their services, once they are performed at the same time.

5.4.5 The Replication Profile

As EJB is a type of implicit middleware, all the profile elements in Table 5.2, except `Replica` are not associated directly to the elements of the EJB programming model, i.e., Home Object, EJB Object and Enterprise Beans (Session and Entity). The other elements presented in Table 5.2 are encapsulated in EJB through the deployment descriptor file.

We realize a Fault Tolerance PSM for EJB with the following elements:

1. `ReplicaHome` - The client never makes requests directly to the EJB beans, but through the EJB objects. Clients get a reference of EJB objects through an EJBHome object, which is responsible for instantiating, finding existing and destroying EJB objects. The `ReplicaHome` carries out the functionality of this factory.

2. `ReplicaSession` and `ReplicaEntity` - The classes stereotyped as such implement the `Replica` class from the Replication Reference Model (Figure 5.12) and the functionalities of the EJB Session and EJB Entity Beans, respectively. As a type of `Replica`, the classes having these stereotypes will be monitored by the object group that implements the `Manager` functionalities.

3. `EJBManager` which implements the functionalities of the `Manager` required to monitor the behavior of the replica objects. In JBoss, each partition comprising the `HAPartition` is an EJBManager.

4. `EJBGroupManager` as the set of partitions it manages (Layer 2) in JBoss. Due

to the implicit middleware nature of EJB, the deployment descriptor file declares the service requirements for the EJB components, such as transaction control and partition management. Every resource to be deployed in the EJB Container needs an association with the deployment descriptor file. This information is used by the EJB Container where the EJB components will be deployed. In the context of JBoss, the `DistributedReplicantManager` , the `HASessionState` and the `DistributedState` perform the functionalities of the `GroupManager`.

5. `EJBSessionStyle` applies to Session Beans so that the persistent or transient replication style can be chosen. If persistent mode is chosen as a style, the Session Bean descriptor file requires an element to manage the cluster-wide distributed service. This value is an attribute of `HASessionState`, as part of JBoss architecture, and is specified through the element `<session-state-manager-jndi-name>` which is a sub-element of the `<cluster-config>` element in the JBoss descriptor file.

On Table 5.3 we summarize the elements that constitute the profile we built for the EJB Profile.

EJB PSM Elements	*FT Profile Elements*	*Tag*
ReplicaHome	Replica	–
ReplicaSession	Replica	Jndi-name
ReplicaEntity	Replica	–
EJBGroupManager	GroupManager	–
EJBManager	Manager	–
EJBSessionStyle	ReplicationStyle	–

Table 5.3: Profile Elements for Fault Tolerance in EJB.

After identifying the elements that constitute the replication transparency profile for EJB, we define coarse-grained mapping rules[2] that complement those defined in the PIM model of Section 5.4.1 in order to generate the model tailored for EJB vendors.

Our rules were defined based on our experience and the documentation about clustering from JBoss [LB04] (chapter 5 in particular), as it is the only available free implementation of fault-tolerant EJB with respect to replication.[3] The rules are mostly configured in the deployment descriptor files (*jboss.xml* and *server.xml*) in the stereotyped `relDeploymentSpec`

[2]These rules may vary according to the J2EE vendor, but we aim for, as much as possible, a generic EJB vendor.

[3]We also point out here this mapping targets the clustering of EJB Beans as a replica and not particular fault-tolerance services or code.

defined in Section 5.4.1. However, we define here rules more specific to the EJB elements, which are the following :

Rule 1 The `ReplicaSession` or the `ReplicaEntity` inherent the attribute `ReplicaID` from the parent `Replica`. In JBoss, the `ReplicaID` is mapped to the Bean name itself, `<ejb-name>` attribute in the *JBoss.xml* file. We define this rule under the metaclass stereotyped `relDeploymentSpec`.

Rule 2 The `EJBGroupManager` (which maps to the properties similar to `<cluster-config>` element in the *JBoss.xml* descriptor file) inherents from the `GroupManager` two major tag definitions: (1) the `EJBManager` identifier, inherited from `Manager` identifier (`ManagerID`), as the name of the partition (`<partition-name>` attribute in `<cluster-config>`), (2) the policy from the relationship with `PolicyManager`, which can be the first available node or round robin, applied to `<home-load-balance-policy>` and `<bean-load-balance-policy>` in `<cluster-config>`.

Rule 3 If `TransientStyle` of the `ReplicaSession` class element, is `Stateful`, the descriptor requires a JNDI name, `<session-state-manager-jndi-name>`, initialized as Default.

We formalize the above rules in OCL in order (1) to keep the consistency between PIMs and PSMs, and (2) to avoid ambiguities between the models. Those rules are expressed in OCL in Appendix A.

5.5 Related Work

Using UML profiles to support modeling of non-functional aspects of software systems following a model-driven approach is not a new idea. The approaches for model-driven non-functional analysis are distinguished mostly by the way they support analysis of annotated UML models.

There has been much previous work dealing with reliability at the design level [CSC02, HM01, SCC+01]. Those works look at some of the issues we identify in our work, in terms of addressing dependability concerns in the early stages of software development. We can primarily find in those works analysis techniques to validate design tools based on UML.

Our approach can thus complement those approaches in many aspects. MDA uses the straightforward approach through the concepts of mapping models among different platforms. Adopting the MDA approach, we believe that we can construct an environment where one can

consistently integrate the analysis and design of dependability issues, as well as the integration of design and implementation. [BMM99] provides a useful transformation technique to automate dependability analysis of systems designed through UML. Nevertheless, to properly contemplate dependability in all stages of the software engineering process, we believe that one of the main concerns is to provide a unified semantics among the analysis and the design models. [MPB03] has presented a more elaborate work of how UML 1.4 can be applied to stochastic analysis of system architecture dependability.

Another approach to address software dependability is to provide mechanisms to improve reliability of software after it has been implemented through testing techniques. Works such as [FHLS98] use those techniques to identify faults in the software that are likely to cause failures. Testing for reliability carries out an important research agenda, and is an important tool to measure the reliability of software systems. In our work, we first assume that the reliability property of the system components will be already provided most likely through testing. Secondly, our purpose is to concentrate on the reliability assurance of the system from design to deployment level through profile support for transformation techniques. Therefore, concentrating on these levels, we believe that the desired reliability of software systems will be reported in the testing phase according to the required reliability property defined in the architecture level.

As mentioned previously in this chapter, the OMG has also requested proposals in order to standardize profiles for particular application areas, in particular the QoS profile. The major concern of the QoS profile are the QoS aspects, including, among others, dependability. However, it can be viewed only as a starting point towards the specification of an MDA profile for dependability. The revised submission does not provide enough details in terms of the relationship between the Dependability sub-profile (as part of the QoS Profile) and the SPT profile, which considerably restricts the inclusion of dependability analysis following a standard MDA approach. A more refined level is required for the QoS profile in order to make use of it as a reference model and in terms of solutions to mitigate reliability problems it seems to be so far restricted.

There has been work following the MDA approach for non-functional requirements modeling by extending the SPT Profile with regard to performance [GP03, SE01]. Gu et al. implement a transformation by parsing the XMI output of profile-mapped UML diagrams [GP03]. The approach of Skene et al. resembles that of Gu et al. but provides a more formal elaboration of the profile via constraints [SE01]. Our approach follows in the same standard-based spirit, but with regard to reliability modeling. At the end of the day, any standard-compliant UML tool is capable of storing these models.

Cortellessa et al. [CP04] proposed an amendment to the QoS Profile [OMG04b] with the purpose of addressing issues related to the reliability modeling of component-based systems. Our profile follows a similar structure as their extension for the QoS Profile, but we differ in the way we compose scenarios. In particular, we consider it important to provide more structure to a scenario specification and thus to model the interaction between scenarios from a global perspective. For instance, using the *HMSC* as part of the profile allows us to model larger systems. Therefore, we believe that our profile provides gains in modularity for modeling large systems and their reliability issues. More recent work from Cortellessa et al. in [CMI07] also presented an integrated framework for reliability analysis following the MDA approach. They describe the work in the COBRA framework, with three different levels of abstraction to incorporate reliability analysis following the MDA approach. Though it is a more recent approach compared to ours, we can fit our approach depicted in Figure 5.1 into theirs where their Computational Independent Non-Functional Model is on the level of abstraction of the Reliability Domain Modelling of that figure, whereas their Platform Independent Non-Functional Model is on the level of abstraction of the Analysis profile of the same Figure 5.1 (further outlined in Figure 5.4) and their Platform Specific Non-Functional would be a further interaction of that Analysis profile considering the Deployment Modelling. One of the major difference on this last aspect though is that in the platform specific level, they consider the failures arise from a continuous time events, while ours is suited to discrete time events, as a consequence of applying Cheung approach for reliability estimation.

In our profile, we use of UML 2.0 constructs to support reliability analysis for component-based software systems. Constructs in UML 2.0 make easier the task of modeling non-functional requirements due to its richer expressiveness compared to previous UML versions. Reliability modeling using new concepts introduced in UML 2.0 are not commonly found in the literature. We believe that wider availability of modelling tools supporting UML 2.0 will stimulate new work in this area.

With regard to model-driven fault tolerance, Majzik et al. provide a profile for modeling fault-tolerant mechanisms, particularly redundancy, in UML diagrams [MPB03]. Transformations are done in a sound manner through graph transformation, from UML to their analysis platform. Approaches in this category do not follow a standardised MDA approach though. As a result, the key benefit of a standards-based approach is lost, i.e., interoperability of applications enabling a market in robust industrial tools that support the approach. But the most expressive is definitely the FT profile in [OMG04b]. The profile is quite comprehensive, following basic concepts of fault tolerance techniques. However, the profile does not provide

integration between the various other profiles that compose the software reliability engineering cycle, as we offer in this chapter. Without integration, the benefits of the automation proposed by MDA cannot be achieved. Furthermore, we also showed through the application we configured and deployed using an EJB vendor which provides fault tolerance, that few other concepts were required in order to fit the structure of a fault tolerant EJB application into the UML profile for fault tolerance in component-based systems.

5.6 Summary

In this chapter we defined UML profiles for achieving reliability in the model-driven engineering context by following the MDA principles. Following the steps to accomplish our approach for model-driven software reliability prediction, we introduced the large box in the upper right-hand side of Figure 5.19. We started with a reliability conceptual model and traversed the elements on that model related to reliability by designing a UML profile for them, where a PIM and a PSM model are defined. In particular, we focused on EJB as a PSM, where UML model elements and rules were defined. Our limited validation for the EJB profile was based on the parallel we made between those profile elements and rules with the well known JBoss EJB vendor. In Section 5.2, we presented a UML profile for reliability analysis methods derived from scenario specifications. As previously mentioned, scenarios are often required in dynamic modeling of requirements in component-based software systems. As a matter of fact, it is natural that the approach of using scenarios for reliability analysis is (or has even greater potential to become) widely adopted by software engineers. Therefore, the profile is meant to enhance the current QoS profile, so that other reliability analysis techniques can benefit from the standard.

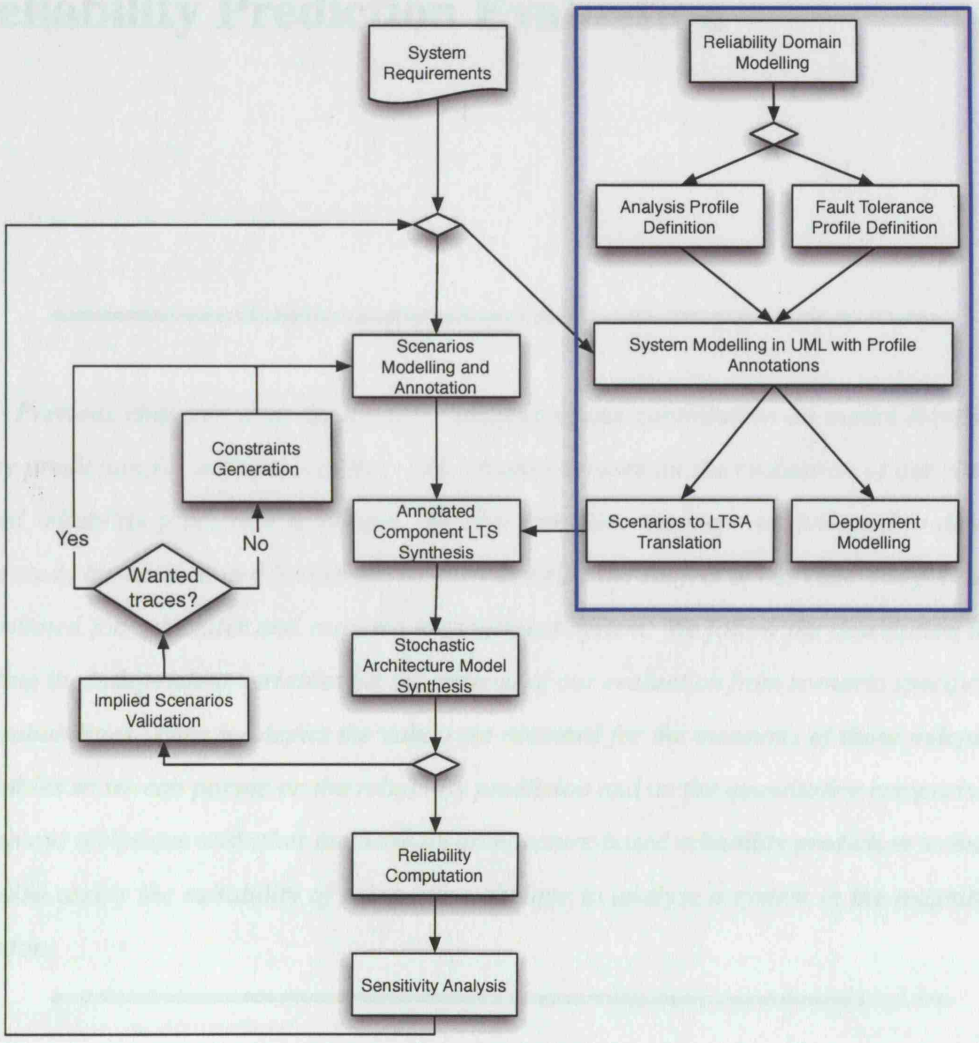


Figure 5.19: The Elements of Model Driven Reliability Prediction Framework Discussed in Chapter 5.

Chapter 6

Reliability Prediction Evaluation

Previous chapters were dedicated to presenting our contribution on model driven reliability prediction for software systems. This chapter focuses on the evaluation of our scenario-based reliability prediction technique. We first introduce the steps we followed to define the case study based on sound literature. We then describe the subject of our case study, Condor, a distributed job scheduler and resource management system. We follow the description by presenting the independent variables for the purpose of our evaluation from scenario specifications to probabilities. Then we depict the values we obtained for the measures of those independent variables so we can pursue on the reliability prediction and on the quantitative comparison between our technique and other mainstream architecture-based reliability prediction techniques. We also assess the suitability of using our technique to analyse a system in the magnitude of Condor.

6.1 Case Study Overview

In this chapter, we present an evaluation of our reliability analysis technique.The case study we present is based on sound literature on empirical studies [PPV00, KPP95]. On the whole, a case study to be complete should be structured into the following seven steps:

1. Research Context
2. Hypotheses
3. Identify the method of comparison
4. Experimental Design
5. Threats to Validity
6. Data Analysis and Presentation
7. Results and Conclusion

The research context has been already explored throughout the thesis. But we still need to characterize the other steps.

Informally, we want to answer two questions regarding our reliability analysis technique: (1) To what extent is the technique suitable to accurately predict the reliability of component-based software early in the development lifecycle? (2) Do the outcome features (e.g., sensitivity analysis) of the analysis hold? Formally, our purpose for carrying out a case study is to evaluate the following set of hypotheses with respect to our reliability analysis technique:

> *Hypothesis 1* – The technique provides reliability predictions early in the software life cycle comparable to the accuracy of that obtained through canonical methods of reliability estimation after development.
>
> *Hypothesis 2* – The technique identifies components and scenario transitions (in case of branching transitions) that have strong impact on the system reliability as a whole.
>
> *Hypothesis 3* – Accounting for implied scenarios improves the accuracy of the predicted reliability of a software system.

To verify hypothesis 1, we need to measure the accuracy of our techinque by comparing our results with those methods we presented in Chapter 2. The accuracy consists in verifying how much our analysis results differ from those obtained by measurement and canonical ways to compute software system reliability. There is one important distinction to be made though: reliability estimation versus reliability prediction. As defined by Musa et al. in [MIO87], *estimation* is the statistical inference procedures applied to failure data taken for the program.

Prediction is the determination from properties of the software product and the development process before any execution of the program. Although our reliability analysis technique targets early reliability prediction, one of our main purposes in the case study is to compare the values obtained from applying our technique and those computed using both a canonical reliability estimation and other prediction techniques proposed in the literature.

Other benefits of applying our reliability analysis technique also consist in identifying (1) those components that impact more on the system's reliability, and (2) the influence of detecting implied scenarios (specially when they identify unwanted software execution traces) on the reliability computation. We want to verify, as stated on hypothesis 2, if the information we derive from our analysis is consistent with that observed experimentally or otherwise analytically, when it is not possible to verify the properties in practice.

We evaluate our reliability analysis technique through a significant case study with Condor, a distributed job scheduler and resource management system [DTL02, Uni06]. This remainder of this chapter is organized as follows: Firstly, we describe in Section 6.2 a design plan with the elements required for the case study following major references on empirical work [PPV00, KPP95]. In Section 6.3 we highlight techniques that we use used to compare our results to other reliability prediction techniques including mainstream architecture-based ones. We describe in Section 6.4 the Condor system, including its architecture and scenarios and their components and in Section 6.5 we cover the independent variables defined in the case study. In Section 6.6, we highlight the dependent variables we aim for the case study. In Section 6.8 identify the threat to validate the conclusions we draw from the case study. In Section 6.7 we present a quantitative analysis of the outcomes of our case study, the comparisons and sensitivity analysis. We conclude the chapter with a critical evaluation in Section 6.9.

6.2 The Case Study Design

In order to test the hypotheses, we need to set up the structure of the case study. The primary components are the variables linking cause and effect. Dependent variables are those expected to vary when attributes, i.e. independent variables, defining the study setting change. Before we introduce our case study, we describe in general terms the basis of our evaluation. It comprises:

Independent Variables – The independent variables of the case study are: (1) *Scenarios* describing interactions between components, (2) *Probabilities of scenario transitions* and (3) *Component Reliabilities*. Scenarios are obtained from system documentation corroborated by expert knowledge of the system. Probability of transitions between scenarios are estimated from logs that collect system usage information. And lastly, the reliability for

each component with respect to correct message processing is estimated from logs that collect error messages when either the components were unresponsive or their hosting machines were down. If component reliability is not provided explicitly by component developers, it is important for the components to have a clear interface, so that system failures can be properly categorized and associated to their respective components. The metric for the components reliability required for our technique is represented in terms of successful execution of component messages on a single system execution. If this is not possible to be obtained directly, other means that are comparable with that metric need to be employed.

Dependent Variables – To obtain the dependent variables, i.e. those output values that we use to verify our hypotheses, we need to measure the *accuracy* and the *validity of the output* produced by applying our technique. In addition to the reliability quantification, we also perform *sensitivity analysis for component reliabilities and transitions probabilities*, identifying those components and scenario transitions that play an important role in the reliability computation. Another dependent variable involved in the case study is the *reliability computation accounting for implied scenarios*. In Section 6.6, we present the study design to obtain various measures for the dependent variables by systematically manipulating the independent variables. Other analysis of the dependent variables we can apply is derived from the case study, and depends on the nature of the system and the failure classification.

Risks Involved – In a component-based reliability analysis technique, it is usually a difficult task to obtain the reliability values for the components from scratch. One of the reasons for that is the need to clearly associate failures with a particular component. It is common practice to obtain software faults as done in bugtrack reports such as those provided by Bugzilla [Bug07]. However, tracing those faults to their components is usually done in ad hoc ways with no clear association to the failure and the components those faults relate to. Also, it is not always present in the fault description information about the environment settings where those failures were revealed. Were that information provided, this would help the characterization of failures so that it would be possible for the quantification of software reliability into appropriate metrics. For those reasons, we consider a risk to our evaluation the extraction of component reliability, specially due to the way our technique requires the component reliability to be expressed in terms of message executions. For this purpose we need to have enough information to obtain that quantification by mon-

itoring components or converting execution time into a number of message executions. As a result, this case study is also verifying the feasibility of applying our technique on a real life software system not initially planned to provide us the input variables in the format we require. Section 6.8 presents a complete discussion of the threats to validity of our case study.

6.3 Comparison Methods

The goal to analyse our technique for its precision consists in comparing the results obtained through our technique with those obtained via canonical ways of computing software systems reliability. Canonical ways to estimate the reliability of a system can vary depending of the type of data available. In a component-based software reliability model, if the configuration of the software components can be assumed to have independent failures, the software reliability can be calculated following the simple AND-OR composition of component reliabilities [MIO87]. If *all* components must function successfully (AND configuration), or if the failure of the software will cause the failure of the system, the system reliability can be computed as :

$$R = \prod_{i=1}^{n}(R_{Ci}), \tag{6.1}$$

where R_{Ci} is the reliability of a component Ci and n is the number of components the system comprises.

The other alternative is the OR configuration, where successful execution of *any* component will result in system success. This is usually the case for component replication. The computation of such a configuration is as follows:

$$R = 1 - \prod_{i=1}^{n}(1 - R_{Ci}) \tag{6.2}$$

The verification of the accuracy of our technique consists in comparing the output obtained by applying our reliability analysis technique with canonical techniques to estimate software reliability.

Additionally, there are various approaches described in the literature for software reliability prediction [GPT01]. Our choice is to compare to a representative example of each model type: (1) Path-based: Yacoub et. al [YCA99], (2) Composite state-based: Cheung [Che80], and (3) Hierarchical state-based: a variation of Kubat's model [Kub89], presented in [GPMT01]. Yacoub et al. provide the Component Dependency Graph (CDG) approach, which they claim to be easier to implement than a complex analytical model.

From the state-based models, we choose one of each type: a *composite* and a *hierarchical* one. The composite model combines the architecture of the software with the failure behaviour into a composite model used to predict the system reliability. On the other hand, the hierarchical model first solves the architecture model and then superimposes the failure behaviour on the architecture model to predict the system reliability. As a composite model, we chose the full approach of Cheung, whose prediction formula was already introduced previously in our work. We use Cheung's result to compare to ours, which considers the concurrent nature of the system. As for the hierarchical state-based model we chose Kubat's variation, whose use is facilitated in our case for it uses the same input of Cheung [Che80]. Rather than choosing one approach only, we choose to apply the three of them (a path-based, a composite and a hierarchical) in order to obtain a good range of comparisons, which are the same models used by Goševa-Popstojanova et al. in [GPMT01] and in [GPHP05].

6.4 The Condor System

The aim to conduct this case study is to verify the hypotheses we have specified in the beginning of this chapter on a real system. That includes comparing the analysis results we obtain by applying our analysis technique on a real-life case study and comparing the results with those obtained using a canonical methods of reliability estimation.

The system we used as a case study is a highly available Grid system called Condor [DTL02, Uni06] distributedly running at some departments across University College London (UCL) [CWT+04]. Condor can be characterized as a distributed job scheduler and resource management system that uses various fault tolerance mechanisms to guarantee very reliable execution of scheduled jobs. We provide more details about Condor and the case study itself in the next sections.

In addition to verifying our hypotheses when modelling systems of the magnitude of Condor, we also want to verify the feasibility, as well as the effort required, in obtaining the independent variables to apply our analysis technique.

6.4.1 Condor Architecture & Components

Condor provides means for users to remotely submit jobs as executable programs in a distributed manner. Additionally, based on a set of parameters (e.g., job requirements, resource ownership and workload policies) Condor manages the execution of those jobs on computational resources (henceforth, referred to simply as resources) selected from a pool.

The primary tasks performed by Condor are: resource allocation to jobs, job startup and execution (after which Condor returns results to clients), and collection and display of meta-

data. As these tasks happen separately within the Condor architecture, Condor components can be clearly defined. Figure 6.1 shows how the decomposition of the Condor architecture is distributed in three major servers: Central Manager, Submission and Execution. There can be only one Central Manager, but there can be any number of Submit and Execution machines. There is one main Submit machine and other project-specific ones, while there are around 1400 Execution machines, in the *Condor pool* running across some UCL departments. The UCL Condor pool is a distributed high-throughput cluster of approximately 1400 Windows PCs, ideal for running large numbers of similar jobs. The Condor pool uses the Condor resource management system developed by the University of Wisconsin-Madison. Between October 2003 and July 2005, UCL Condor managed, processed and returned results close to 2,000,000 hours of execution of computational chemistry, physics and geological programs for users all over the UK academic community, a total that is roughly equivalent to 220 years of CPU time [UCL07].

The major coarse-grained architecture description constitutes of the following elements:

Central Manager – Collects information about the Condor pool and matches job requirements to machines that are suitable to the community policies (job pre-emptive rules and priorities). It hosts two *daemons*: *Collector* and *Negotiator*. The Collector gathers information from job characterizations through *ClassAds* in order to enable one to determine the state of all machines in a Condor pool. The *Negotiator* then takes this information and does the matchmaking of users and resources. The *Negotiator* can also manage the user priorities and job preemption rules to identify the semantics of the service provided by a pool.

Submit – Allows users to submit a job to a local and persistent queue, which is maintained by the *Schedd daemon*. This daemon advertises to the *Collector* the number of idle jobs on the queue and then submits requirements for its jobs in priority order so as to make the suitable resource allocation. After the user-resource match is done, the *Schedd* starts a *Shadow daemon* which manages the remote execution of the job. The *Shadow* then interacts with and monitors the *Starter* on the remote machine. Additionally, it manages the checkpointing of a job, surrogates the remote application and reschedules the job if it fails.

Execution – Runs jobs on behalf of clients and has the capability of advertising its capabilities and usage information. It hosts the *Startd daemon*, which represents a resource in the Condor pool. When the match between the Central Manager and the Submit occurs, the *Startd* will spawn a *Starter daemon* responsible for managing the local execution job. The *Starter* initiates the actual job on the execute machine, sets up the execution environment,

monitors the progress of the job and cleans up the execute machine on job completion.

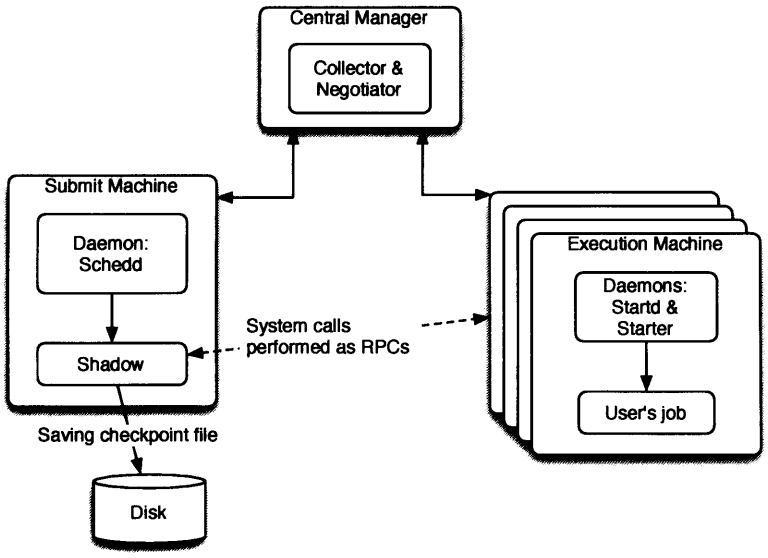


Figure 6.1: Condor Architecture Overview [Uni06]

In the context of the case study, the deployment scenario is one where each of its major architecture elements, i.e. *Central Manager, Submit* and *Execution*, are deployed on different machines. For the sake of simplicity we mention the *Execution* element, but in fact, it represents a pool of machines where jobs will be executed.

6.5 Independent Variables for the Condor Study

As pointed out in Section 6.2, our case study has three independent variables: (1) scenario specifications; (2) reliability of the components, and (3) transition probabilities on branching between scenarios. In the context of Condor, three other variables are part of the set of independent variables:

1. job submissions and their properties, i.e. the number and time duration;
2. the class of failure for the components;
3. the number of machines (or resources, equivalently) available to execute the jobs submitted to Condor; and
4. the failover overhead of Condor.

The statistics from which the analysis of Condor was derived are the following:

- Number of jobs for the submission/cancellation analysis: 128058 jobs.

- For the availability of the resource, data was extracted every ten minutes over two different periods of time. The initial period was from July to August 2006 while the second measure was obtained during that same period in the following year.

- The logs available for the information regarding the Central Manager and the Submit machines were firstly obtained from the period May to August 2006, and secondly from the period October 2006 to June 2007.

- The information related to the failover of the jobs was collected from the period July 2007.

At a first glance the different periods where these log data come from seem to be random. But in fact, the way those data are obtained from Condor is done in different ways. Each component in Condor is independently monitored and collected in different periods of time. We read data from Condor logs during two periods: one in the period of end of July/ early August 2006 and a second experiment during the period of end of July/ early August 2007. Our first attempt once we decided to use the instance of Condor at UCL was to make use of the information at hand. By that time there was no logged information for the availability of resources under a regular basis. That required the administrator to set up a new script to log that information. This is the reason why the period of data obtained for the resources availability is quite different from the period of data for the Central Manager and the Submit machines. On top of that, the disruption of one year time was caused by the fact the log could not be kept for a long time due to the amount of information it was gathering (i.e resource availability every 10 minutes).

Regarding the information for the failover, in the first experiment we did not consider the failover mechanisms of Condor as those data were not available and our knowledge about the system was still incipient. After some learning about the system and extensive discussions with the Condor administrator at the moment in UCL, we managed to find ways to model more accurately the availability of the resources in Condor including information from the job failovers.

6.5.1 Condor Scenarios

To present the scenarios that constitute Condor, we describe first the functionalities from where the scenario specifications will be derived. Condor has the following major functionalities:

- Resource management – A central manager in Condor collects resource characteristics and usage information from machines in a pool of resources, the Condor pool. From the information collected, such as the underlying availability of resources, jobs can then

be scheduled with suitable resources. By this means, users can estimate the resource potential before submitting job to the schedulers.

- Job submission – Remote clients submit jobs with input files and executables. Users can configure their job submission by assigning different priorities to different jobs and specifying workflow dependency between jobs.
- Job scheduling and allocation – A scheduler in Condor manages requests for job executions from various users' submissions as well as maintaining a persistent queue of jobs.
- Job execution management – Remote users can manage their jobs once submitted, including job monitoring and execution control as well as cancellation of submitted jobs either before or after those jobs are allocated to a specific resource. Condor provides mechanisms to manage remote execution of jobs such as *file transfer*, to place files on remote machines; *checkpoint*, where jobs state can saved and subsequently resumed; and *migration of jobs* among machines.

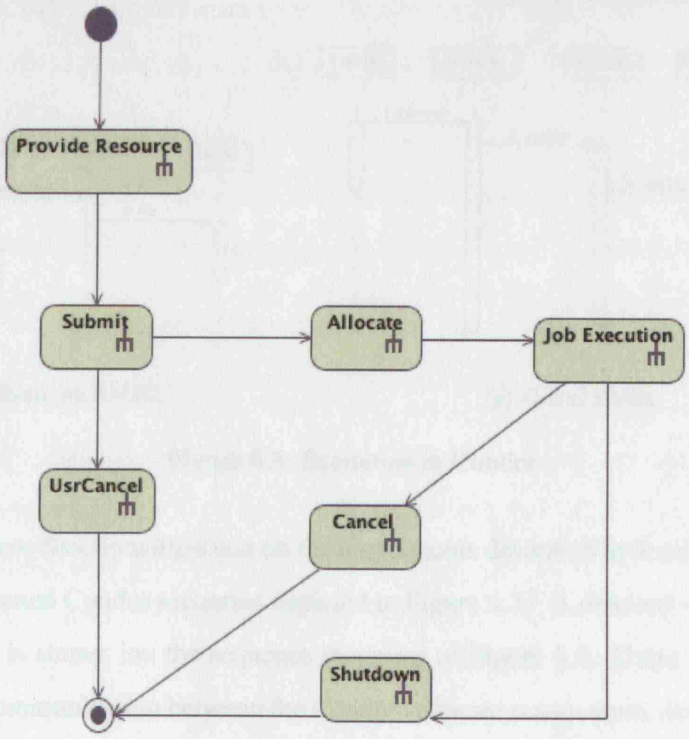
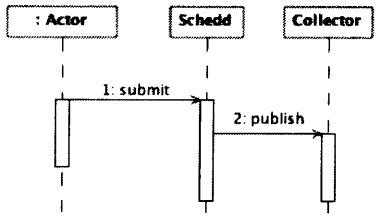
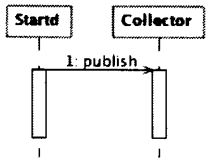


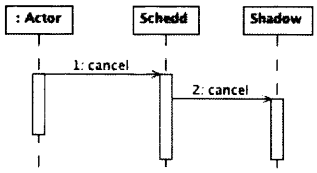
Figure 6.2: High level scenarios of Condor (HMSC)



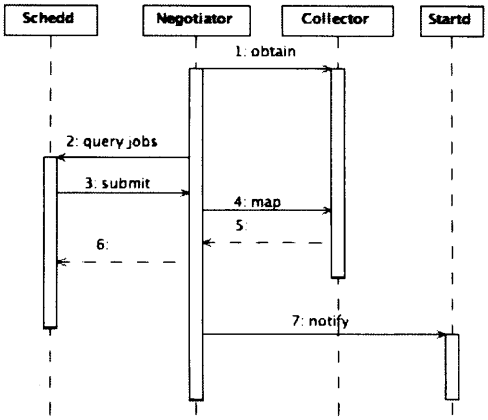
(a) Submit BMSC



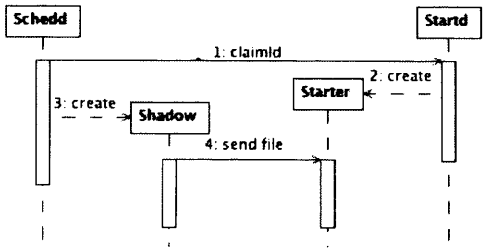
(b) Provide Resource BMSC



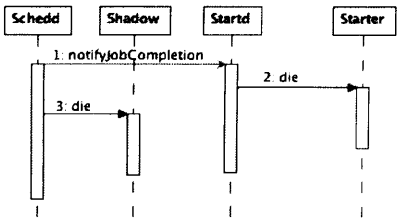
(c) UsrCancel BMSC



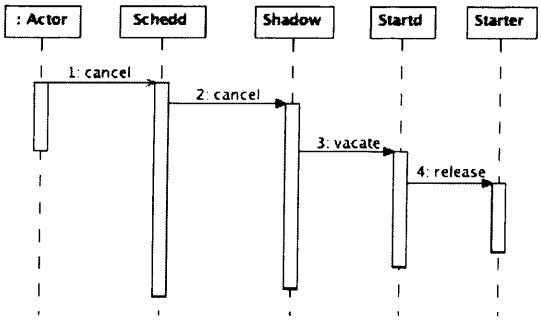
(d) Allocate BMSC



(e) Job Execution BMSC



(f) Shutdown BMSC



(g) Cancel BMSC

Figure 6.3: Scenarios in Condor

Based on these functionalities and on the components described in Section 6.4.1, we specify the coarse-grained Condor scenarios depicted in Figure 6.2.[1] A detailed view of the scenarios of Figure 6.2 is shown inn the sequence diagrams of Figure 6.3. Those diagrams model a coarse-grained communication between the Condor software components described previously in Section 6.4.1.

It is important to note that, although Condor implements various mechanisms for fault tolerance, we do not explicitly model the effects of replication, since this would lead to state

[1]The mark similar to 'ⓜ' in the nodes of the Activity Diagram is a notation in the Magic Draw UML tool we modelled the scenarios to indicate there is a hyperlink between the nodes and their respective Sequence Diagrams.

explosion in the analysis model. As a result, this limits the ability to compare predicted reliability to measured one, which is why we opt instead to compare to other techniques in Section 6.7.

6.5.2 Scenario Transition Probabilities

The reliability measurement provided by our technique requires an environment setting where we can obtain the occurrence probability of scenarios. These probabilities were obtained from system logs of usage measurement and correspond to operational profiles, which are typically obtained from similar previous systems (as we have simulated here), or otherwise estimated by stakeholders.

In order to specify transition probabilities between branching scenarios we need to determine the frequency with which the scenarios are executed. From this information we derive the *transition probability* of choices between scenarios as well as the *frequency with which components are invoked.* In particular, from Figure 6.2 we can that the scenario transitions of interest to be monitored are:

1. from scenario *Submit* to scenarios *Allocate* and *UsrCancel*, and
2. from scenario *JobExecution* to scenarios *Cancel* and *Shutdown* (the latter representing successful job execution).

All other scenario transition probabilities are 1.

The information for the occurrence probability of scenarios in Condor is obtained from the job submissions log. This log collects information from jobs submitted by all Condor users. From the excerpt in Table 6.1, it can be seen that there are three main pieces of information we can use to obtain the desired probabilities: (1) the status of the job (column `ST`), which can be either completed (`C`) or cancelled (`X`), (2) the date the job was submitted, (3) the job duration, and (4) the total number of jobs submitted per month. We implemented a simple Java program to filter out this information.

ID	OWNER	SUBMITTED	RUN TIME	ST	COMPLETED	CMD
47.0	user 1	1/11 08:08	1+02:23:40	C	1/12 10:32	site 1
48.0	user 2	1/23 07:31	0+00:00:00	C	1/23 07:31	site 2
75.4	user 3	1/11 15:35	0+19:30:02	X	???	site 3
78.0	user 4	1/31 12:49	0+00:00:00	X	???	site 4

Table 6.1: Job History Queue Excerpt

In order to obtain the branching probabilities we distinguished those jobs that were cancelled in two different ways: those with run-time zero (0+00:00:00) and those with run-time

greater than zero. If the job is cancelled with run time zero (0+00:00:00), we interpret this as meaning that the scenario *UsrCancel* was triggered by the user. Otherwise, with. cancelled job run-time greater than zero, we can interpret this as meaning that cancellation happened after job was allocated and thus, scenario *Cancel* was triggered. Usually, this latter cancellation is operated by the administrator. The frequency of transition to the *Allocate* scenario is therefore obtained from the total number of job submissions minus the number of jobs cancelled with run-time zero. And the frequency of transition to scenario *Shutdown* is obtained from the total number of non-user-cancelled jobs minus the number of jobs with cancellation time greater than zero. The results we computed are presented in Table 6.1 an excerpt from the job submissions log.

Transition	*Probability*
Submit → Allocate	94%
Submit → CancelJob	6%
JobExecution → CancelJob	3%
JobExecution → final	97%

Table 6.2: Scenario Transition Probabilities for Branching Scenario

6.5.3 Component Reliabilities

One of the most important independent variables, as well as the one which proved most challenging to obtain, is the reliability of the components. The major reason is because component failures are not always clearly classified in log data. If that classification is not clear, the may deem unfeasible the reliability estimation of a component-based system. For this reason, we first need to identify the class of failure we are aiming at for Condor components. After this, we describe how we quantify the reliability of Condor components.

The failures we are looking for in the Condor components are those where they are unavailable or unresponsive to message invocations. As Condor implements various fault tolerant mechanisms with a highly available and highly redundant infrastructure, jobs hardly ever fail. Unless the Submit machine is not available so that the user cannot submit any job, Condor can almost always guarantee the execution of a job, once submitted. Most of the unavailability of the components are perceived in Condor as their downtime or the downtime of the machine hosting the component (Condor-Master). In order to apply this in our reliability prediction technique, we need to associate the unavailability as failure in terms of message invocation. From a general perspective, if we uniformly distribute the component unresponsive time over an interval, expecting that time to be roughly equally spaced, we can define the component

unavailability in terms of message invocation failure. This can realized by the simple formula below:

$$R_{msg} = 1 - \frac{f_{msg}}{n_{msg}} \quad (6.3)$$

$$= 1 - \frac{(n_{msg} * U_{comp})}{n_{msg}} \quad (6.4)$$

where:

- f_{msg} is the failure of message invocations to the component, in this case, the number of messages invoked on the component while it was unavailable,
- n_{msg} is the total number of message invocations on the component. It is measured as the number of message invocation to the component per job execution or as the number of times a task is scheduled to happen, and
- U_{comp} is the period during which the component was unavailable.

Therefore, we get the failure of components message invocations based on the unavailability of the component. Taking the unavailability as the percentage of time the component is down over the total monitored time, we get:

$$R_{msg} = 1 - \frac{dt_{comp}}{T} \quad (6.5)$$

where: dt_{comp} is the downtime of the component and T is the total time of the monitoring.

We classify the component failure messages of the Condor log into two major patterns: one related to the component itself (pattern 1) and another related to the machines where the component is deployed (pattern 2). Pattern 1 applies to the four components, i.e, Collector, Negotiator, Schedd and Shadow. Message pattern 2 occurs in the log at both the Central Manager and the Submit machines, which are referred to as the *CONDOR-MASTER* in the logs. We depict below an excerpt from the log messages describing the failure of the components and their hosting machines:

– Pattern 1 related to the failure of the component Schedd:

```
6/13 17:55:51 The SCHEDD (pid 14484) died
6/13 17:55:51 restarting /usr/local/condor/sbin/condor-schedd in 10 seconds
6/13 17:56:01 Started DaemonCore process "/usr/local/condor/sbin
              condor-schedd", pid and pgroup = 21544
```

– Pattern 2 related to the Central Manager and Submit machine restarting:

```
5/12 17:41:40 ERROR "Can't get lock on "/home/condor/log/InstanceLock""
5/12 17:43:25 ******************************************************
7/17 10:43:34 ** condor-master (CONDOR-MASTER) STARTING UP
7/17 10:43:34 ** /usr/local/condor/sbin/condor-master
7/17 10:43:34 ** $CondorVersion: 6.6.7 Oct 11 2004 $
7/17 10:43:34 ** $CondorPlatform: I386-LINUX-RH9 $
7/17 10:43:34 ** PID = 3226
```

From the excerpts above, we can see that we must filter out log information for each component corresponding to two different kinds of time periods. We extract the following four main pieces of information: (1) date and time the failure happened; (2) the duration of the component unavailable time; (3) the initial time of the log and (4) the description of the error.

We do not consider the error messages related to the components deployed on the Execution machines, i.e. Startd and Starter because this information is derived from queries to the ClassAds that regularly extract information from the Execution machines. The queries simply report from time to time information about machines in the Condor pool. It is worth noting at this point that isolating failure data for the Starter and Startd for a single machine is not possible at the log register as the failure message does not relate a failure of Startd on its hosting machine, which would not allow us to obtain the information the amount of time a Startd hosted in a certain machine was unavailable.

Because of the lack of information to obtain individual reliability values for Starter and Startd, we realized the information we were gathering for the reliability of Starter and Startd was essentially about the cluster node as a whole and not a separated information about the Starter and the Startd. Therefore, to correspond to a scenario view consistent with the information obtained from the system, we encapsulate the *Startd* and *Starter*, as a pseudo-component of the Condor pool, namely *Cluster*. From this point on, we replace those components by Cluster on the scenarios.

In order to obtain a measure of reliability for the individual Cluster reliability , we firstly considered the reliability of the execution machine itself whether or not the machine was executing a job. For the second, more recent measure of the Cluster reliability, we considered their reliability as a measure of the rate of job failover as well, as this measure was not available when the first measurement period occurred. The failover indicates a failure happened while the job was in execution. This means that for a job to be successfully executed, it needs the machine resource to be available and that the resource does not fail during the execution. The reliability of Cluster is then computed as follows:

$$R_{msg} = A_{machine} \times (1 - E_{exec}) \tag{6.6}$$

where $A_{machine}$ is the availability of the execution machine and E_{exec} is the percentage of jobs in the execution machines running in the Condor pool that failover to another machine at least once.

In Table 6.3 we present the results for the component reliabilities we obtained corresponding to two periods of time. The data in the column *First Measure* relates to the period from May to August 2006 and the column *Second Measure* to the period from October 2006 to June 2007. Though the values of the components hosted by the same machine present the same reliability, their measure differed in general by 0.001% that cannot be perceived after those values are rounded. Therefore, the major cause for the reported unreliability of the components arise from the unavailability of the machines where those components are hosted.

Component	*May – August 2006*	*October – June 2007*
Collector	97.52 %	99.99 %
Negotiator	97.52 %	99.99 %
Schedd	98.83 %	96.83%
Shadow	98.83 %	96.83 %
Cluster	62.57%	63.63%

Table 6.3: Component reliabilities

6.6 Dependent Variables for Condor Study

Having obtained the independent variables, we apply our technique to obtain the dependent variables and verify our hypotheses. For this purpose, we need to measure the accuracy and the validity of the predictions and regarding the sensitivity analysis for component reliabilities, scenario transitions and implied scenarios by applying our technique. All in all, the dependent variables we want to analyse from the Condor case study are the following:

The Prediction Accuracy – The verification of the accuracy of our technique consists of comparing the output obtained by applying our reliability analysis technique with those described in Section 6.3. For those approaches, the independent variables required are the failures per wait time, component reliabilities and/or the transition probabilities between states which can be derived from the independent variables of our case study. So there is no extra cost in gathering information to use those models to predicted the reliability of Condor.

Accounting for Implied Scenarios – One of the major distinctions between our technique and other approaches is the ability to take into account unwanted mismatches between required software behaviour and its architecture by means of the detection of implied scenarios. Those mismatches are revealed as a consequence of specifying systems behaviour from a global perspective, while in fact, the behaviour is provided by components, which have a local view of the system. We investigate the presence of implied scenarios in the architecture models. We synthesize from the Condor scenarios and verify if or how they affect the reliability of Condor.

The Sensitivity Analysis – We carry out sensitivity analysis for two variables: (1) the component reliabilities and (2) the scenario transition probabilities. The analysis for the former consists of computing the system reliability as a function of varying individual component reliabilities with the purpose of identifying components that have the greatest impact on the reliability of the software system. The method consists of varying the reliability of one component at a time and fixing the others to their original values. The analysis for the transition probabilities consists of computing the system reliability as a function of varying individual scenario transition probabilities. Considering scenarios with multiple outgoing transitions, we want to find out if scenario transition probabilities have a significant influence on our reliability predictions. In case that influence is significant, we want to find out which of those transitions has the greatest impact on system reliability.

Size of Condor pool – This measurement is obtained in order to analyse the number of machines required in the Condor pool to keep it highly reliable, given some specified reliability target. In the analysis we model the cluster by using formula 6.2 to predict the clustering reliability. We then compare our results with observed failover behavior of jobs and those reliability analysis techniques that accommodates fault tolerant architecture style analysis, such as the approach from Wang et al. [WWC99], the one we refer in the Chapter 2, Section 2.2.3.2 as an adaptation of Cheung technique to various architecture styles (pipeline, fault tolerance, etc). In our case, we apply their extension of Cheung work to fault tolerance.

6.7 The Quantitative Analysis

The quantitative analysis is accomplished in three stages: In a first stage, we want to compare predicted reliability results we estimate with those estimated through other approaches, where no fault tolerance is considered and thus assuming there is only one node to execute a job. To do

this, we use the models in Figures 6.26.3[2], where there is only one pseudo-component Cluster available to execute a job. This measure is expected to provide us the worst-case reliability compared to the actual highly available infrastructure of Condor.

In the second stage we carry out the sensitivity analysis for component reliability and scenario transition probabilities, iterating our prediction technique over a range of values for both parameters. We benefit from the expertise knowledge of Condor administrator to assess our results.

In the third stage, we want to verify the importance of the cluster in order to answer questions such as: how many machines are needed to make the Condor pool x% reliable? How many nodes are needed to make/keep the system x% reliable? We then compare our results with data obtained from actual job execution in Condor, where the number of machines the job failed over is the parameter of comparison. As mentioned in Section 6.5.1, we do not explicitly model the node communication or the effect of replication in the scenarios as we assume this is done reliably and does not introduce other problems-f safety and liveness properties of the system. We model, instead, the effect the replication following Formula 6.2 as a canonical way to compute the reliability of a replicated component.

6.7.1 Condor Reliability

The reliability of a grid system can be analysed from two perspectives, as presented by Dai et al. in [DXP06] the grid program and the grid service. For the grid program perspective, Condor implements various mechanisms of fault tolerance, including replication, transaction and asynchronous communication management. Once a job is submitted, Condor will guarantee the execution of the job through its fault tolerance mechanisms, unless the user cancels the job execution. Therefore, we can say Condor is virtually 100% reliable to execute a job, as long as there is no time requirement to execute a job. On the other hand, finding out the required number of machines to run a job in Condor with high reliability imposes a more interesting question. We deal with this question in Section 6.7.4.

As for the grid service reliability we look at Condor as a whole, the ability of Condor to receive the request from a user to execute a batch of jobs. The management of a jobs batch is done in the Submit machine. As previously mentioned, once the jobs are in the Submit machine, Condor will use its fault tolerance mechanisms accordingly to prevent a failure from happening and to guarantee the execution of the job. Therefore, the users' perception of the grid service is mostly related to the reliability of the Schedd component in the Submit machine, i.e. 98.83% on the first measure and 96.83% on the second measure, as shown in Table 6.3.

[2]Note however that we have Cluster replacing components Starter and Startd in those figures.

Additionally, the user can perceive the quality of job execution through the job duration in the job queue list. Logs on the job completion/cancellation status, as presented in Table 6.1, show that some jobs take too much time to get executed, sometimes more than a day, and then get cancelled by the user. Those situations can characterize an error from the user, who might have wrongly annotated the parameters required for the job, or in cases where either the Condor pool is overloaded with jobs or the Central Manager is unavailable to allocate resources for the jobs. This last measure was not able to be quantified precisely, and therefore, an estimate for the user exhaustion for a prolonged job execution lies in the vicinity of 0.1% of the cancelled jobs with execution time exceeding 12 hours.

6.7.2 Our Prediction Results

A first analysis of the Condor model in LTSA showed that a potential deadlock could arise from the scenarios. We further investigated that diagnosis and conducted an implied scenario analysis in order to verify if that deadlock alert could be derived from an implied scenario problem, which then showed to be the case indeed. The implied scenario analysis showed the scenario presented in Figure 6.4.

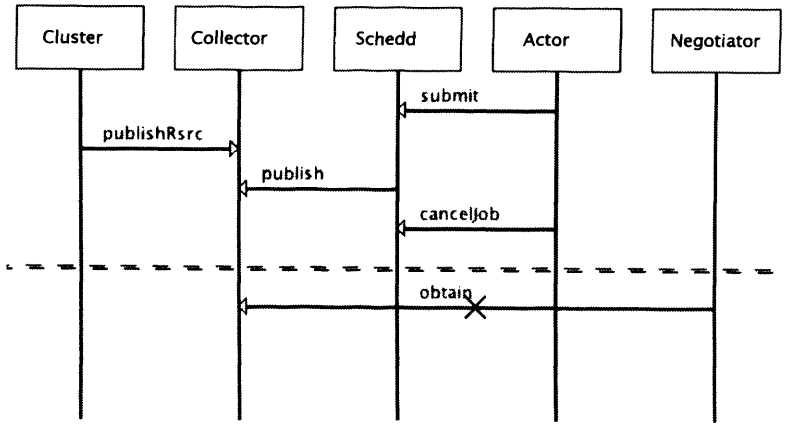


Figure 6.4: Implied Scenario Detected in Condor Scenarios

According to the HMSC in Figure 6.2, the message *obtain* in scenario *Allocate* should only be executed after scenario *Submit*. This means that the *Negotiator* does not have any information about whether scenario *UsrCancel* was executed, and therefore the component *Collector* may receive at any moment an invocation from *Negotiator* to perform *obtain*. If that sequence of scenario happens, i.e. *Submit - ProvideResource - UsrCancel - Allocate*, then this is not modelled in the scenario specification, where after *UsrCancel* there is no possibility that message *obtain* will be executed.

To avoid this mismatch between the intended scenario specification and the resulting anal-

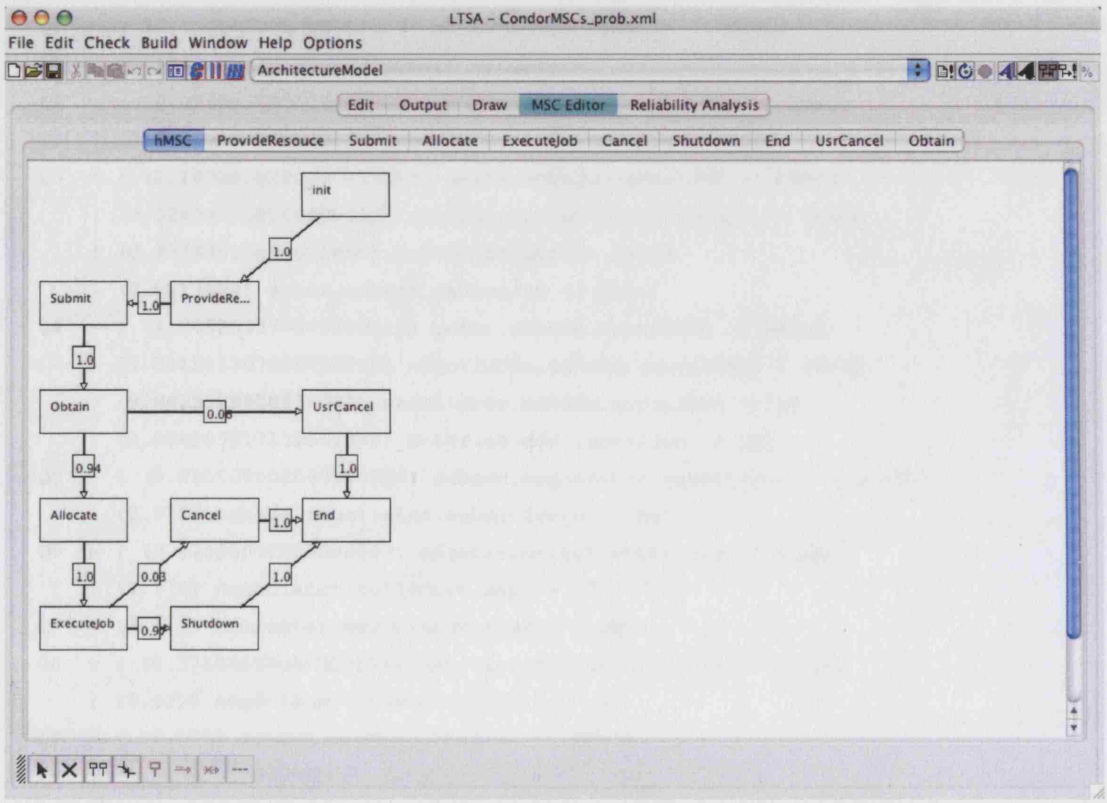


Figure 6.5: LTSA screenshot of the New HMSC of Condor After Refactoring Scenarios

ysis, we can either create a scenario with message *obtain*, which means the identified implied scenario behaviour is acceptable in Condor, or we can make the transition to *Allocate* or *UsrCancel* only after *obtain* is executed. In that case, *Schedd* will either execute *queryJobs* or *cancel*, and it is clear from the scenarios what action should follow if one or the other is prompted. Both situations are acceptable, as the message *obtain* runs asynchronously in practice. However, the approach that suits best the intended Condor scenario specifications is the second one. In the first approach, we would have to model the message *obtain* in two different scenarios: in *Allocate* (in case traces follow the initial specification) and after scenario *UsrCancel* (to accommodate the new traces identified). In the second approach, we only need to put the message *obtain* into a separate scenario, which we call *Obtain* allowing it to happen before either *Allocate* or *UsrCancel*, according to the user's choice.

Listing 6.1: Architecture Model for the Condor

```
ArchitectureModel = Q0,
Q0   = ( (0.006000000000000005) actor.schedd.submit -> ERROR
     | (0.012500000000000008) cluster.collector.publishRsrc -> ERROR
     | (0.4875) cluster.collector.publishRsrc -> Q1
     | (0.494) actor.schedd.submit -> Q20),
```

```
Q1  = ( (0.012000000000000007) actor.schedd.submit -> ERROR
    | (0.988) actor.schedd.submit -> Q2),
Q2  = ( (0.025000000000000022) schedd.collector.publish -> ERROR
    | (0.9750000000000001) schedd.collector.publish -> Q3),
Q3  = ( (2.1600000000000013E-5) actor.schedd.cancelJob -> ERROR
    | (0.024955000000000012) negotiator.collector.obtain -> ERROR
    | (0.973245) negotiator.collector.obtain -> Q4
    | (0.0017784) actor.schedd.cancelJob -> Q18),
Q4  = ( (4.869251577998199E-5) actor.schedd.cancelJob -> ERROR
    | (0.011951307484220026) negotiator.schedd.queryJobs -> ERROR
    | (0.9839909828674481) negotiator.schedd.queryJobs -> Q5
    | (0.004009017132551846) actor.schedd.cancelJob -> Q17),
Q5  = ( (0.025000000000000026) schedd.negotiator.submit2rsrc -> ERROR
    | (0.975) schedd.negotiator.submit2rsrc -> Q6),
Q6  = ( (0.02500000000000003) negotiator.collector.map -> ERROR
    | (0.975) negotiator.collector.map -> Q7),
Q7  = ( (1.0) collector.negotiator.r_map -> Q8),
Q8  = ( (0.37500000000000006) negotiator.cluster.notify -> ERROR
    | (0.625) negotiator.cluster.notify -> Q9),
Q9  = ( (0.375) schedd.cluster.claimID -> ERROR
    | (0.6250000000000001) schedd.cluster.claimID -> Q10),
Q10 = ( (0.012000000000000009) schedd.shadow.createShadow -> ERROR
    | (0.988) schedd.shadow.createShadow -> Q11),
Q11 = ( (0.37499999999999994) shadow.cluster.sndFile -> ERROR
    | (0.625) shadow.cluster.sndFile -> Q12),
Q12 = ( (1.1467402845614789E-5) schedd.shadow.cancelShadow -> ERROR
    | (0.011988532597154399) schedd.shadow.notifyEnd -> ERROR
    | (0.9870558504990443) schedd.shadow.notifyEnd -> Q13
    | (9.441495009556171E-4) schedd.shadow.cancelShadow -> Q16),
Q13 = ( (1.0) schedd.cluster.die -> Q14),
Q14 = ( (1.0) endAction -> Q15),
Q15 = STOP,
Q16 = ( (0.375) shadow.cluster.vacate -> ERROR
    | (0.625) shadow.cluster.vacate -> Q14),
Q17 = ( (1.0) schedd.shadow.cancelShadow -> Q14),
Q18 = ( (0.024250000000000025) negotiator.collector.obtain -> ERROR
    | (0.94575) negotiator.collector.obtain -> Q17
    | (0.030000000000000002) schedd.shadow.cancelShadow -> Q19),
Q19 = ( (0.025000000000000022) negotiator.collector.obtain -> ERROR
    | (0.975) negotiator.collector.obtain -> Q14),
Q20 = ( (0.025000000000000026) cluster.collector.publishRsrc -> ERROR
    | (0.975) cluster.collector.publishRsrc -> Q2).
```

Once we refactored the scenarios and checked the validity of the traces on these new scenarios with Condor's administrator, we synthesized the architecture model for Condor, as presented in Listing 6.1. Computing Condor reliability from that architecture model, under the given deployment environment, we obtained 20.89%. At first, this looks a very low reliability

for such a system like Condor. However, this result is a for a Condor pool containing a *single node* and thus does not take into account the fault tolerance features of Condor, like the failover mechanism available to the jobs. This issue will be approached further in Section 6.7.5.

6.7.3 Comparison

As described in Section 6.3 we computed a reliability prediction for Condor using three other techniques: the path-based technique of using Yacoub et al. [YCA99], the composite state-based of Cheung[3] [Che80], the canonical Musa AND-OR configuration formula [MIO87], and the hierarchical approach of Kubat presented on Chapter 2. Although there are a few other techniques available in the literature using scenarios as well, we considered as much as possible those requiring input information similar to that required in our model. Furthermore, those techniques we chose to compare our results with were also used in previous work of Goševa-Popstojanova [GPMT01][GPHP05]. It is worth noting we assume the independence of failures between the components and do not take into account the failover mechanism of Condor for this comparison.

The first model we consider is the composite one from Cheung, where we start from a state-based view of the system provided by the Condor administrator. The model we constructed together with Condor administrator at UCL is depicted in Figure 6.6. It is worth pointing out the construction of that model does not take into account the MSC scenarios or the concurrent nature of the components. In the next step, we annotate the model with transition probabilities and component (or module in Cheung's terminology) reliabilities. Those are the same input values we used in order to annotate our own scenario specification. As a result, computing the reliability following the Cheung approach does not incur any extra cost in computing model parameters. The computed reliability for Condor following the Cheung approach is 26.373%.

For the hierarchical approach, we use the Kubat model presented previously in Chapter 2 which was also used in [GPMT01] and [GPHP05] by Goševa et al. Kubat model makes possible the comparison of the results as it is adapted to use the same input data as the composite model of Cheung, following the architecture of Condor in Figure 6.6. Therefore, it is adapted to use a state-based model with Markovian properties. The hierarchical formula used to compute the Condor reliability is shown in Formula 2.7. The major difference from the hierarchical to the composite model is that the former does not include the transitions to the absorbing faulty state in a state model.

[3]Notice that although we also use Cheung's formula in our reliability analysis technique, by applying the full Cheung technique we model Condor in the manner required by the technique and thus do not use our own scenarios or models.

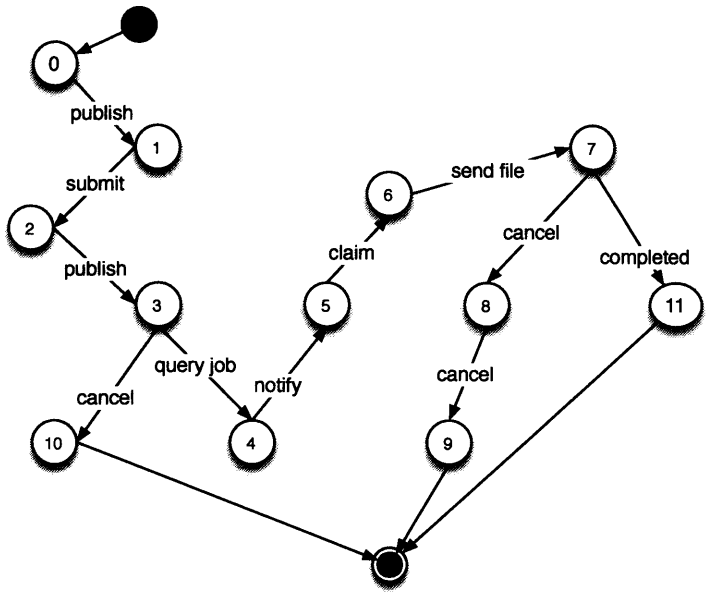


Figure 6.6: Condor Control Flow Graph for the State-Based Approaches

As a result, the number of visits V_i for components per machine is as follows: the Central machine = $2 + (0.94 \times (0.03))$; the Submit machine = $1 + (0.06) + (0.94 \times (1 + 0.97 + 0.03))$; the Execution machine = 3×0.94. Computing Kubat's Formula 2.7 transcribed below, we then obtain 24.33% for Condor reliability.

$$R(k) \approx \prod_{i=1}^{n} [R_i(k)]^{V_i(k)} \tag{6.7}$$

Finally, to apply the approach of Yacoub et al. we constructed a Component Dependency Graph (CDG) as presented in Chapter 2, from our own scenario specifications. The CDG model is depicted in Figure 6.7. However, we encountered the weakness of the Yacoub et al. approach described in Section 4.4, in coming up with the final structure of the model. In particular, unless we take into account implicit transitions between components, it would not be possible to reach the end state of the CDG from the start state.

One example is the transition from Collector to Negotiator. From the scenario specifications, there is no direct message communication between the two of them, which induces a deadlock in the CDG model. In those circumstances, components like Schedd, Negotiator and Shadow would never get the chance to be invoked, and therefore the CDG embodies an inaccurate representation of Condor execution paths. If we compute Condor's reliability via the Yacoub et al. method through the inaccurate CDG, which does not include the extra transitions, the outcome is 75.4%.

In order to make feasible the invocations of all Condor components, we constructed a

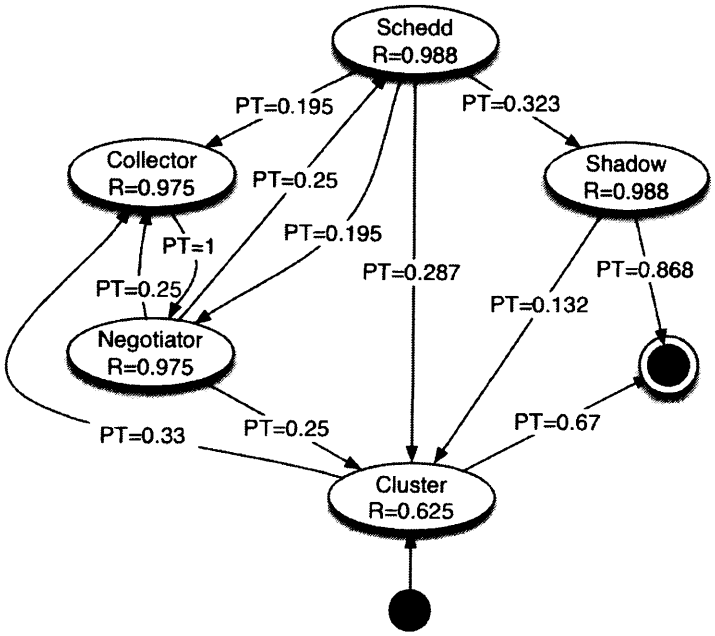


Figure 6.7: Condor CDG Model for the Approach of Yacoub et al.

Prediction Technique	*Predicted Value*	*Difference to Our Prediction*
Ours	20.89%	–
Musa	21.60%	3.39%
Hierachical(Kubat)	24.33%	16.47%
Composite(Cheung)	26.34%	26.09%
Path(Yacoub et al.)	44.13% or 75.4%	111.25% or 260.94%

Table 6.4: Comparison between Reliability Analysis Techniques

second CDG with a transition from Collector to Negotiator, which is present on transitions from scenario Submit to Obtain and from scenario Obtain to Execute Job. Having done this, the computed reliability of Condor, following this modified version of Yacoub et al. is 44.13%.

Table 6.4 summarizes the results of our comparisons. We note that the results for this stage of the case study that our model produces results within roughly 25% of those produced by other reliability models, except the path-based one from Yacoub et al. , which revealed problems during the construction of its CDG, as described previously in this section.

In contrast, the comparison work of architecture-based work of Goševa et al. in [GPMT01] shows a small distance between predictions produced from composite, hierarchical and path based models, although we wonder if the limited number of states (i.e. three, excluding the absorbing states) in their case study precluded a thorough evaluation. In a more recent work, Goševa-Popstojanova, Hamill and Perugrupali in [GPHP05] presented another case study with

a larger number of components (thirteen) but including at that time only composite and hierarchical models in the comparison. Their results once again produced very similar values for both models, i.e. the composite and hierarchical. However, they mention that "the approximate solution of the hierarchical method might not be always so close to the exact solution of the composite model", demonstrated through a hypothesized example they presented in [GPMT01], specially in cases where the system architecture presents loops. Another reason for the low reliability values presented in Table 6.4 can also be attributed to less reliable components (such as the *Cluster*) that are executed many times during a single application execution, also stated by Goševa et al. in [GPHP05].

6.7.4 The Sensitivity Analysis

Considering that we are dealing with a component-based software system, the increase in the reliability of some components often results in an improvement on the system reliability. Through the sensitivity analysis we can identify important components whose reliability improvement produces higher impact on the increase of the system reliability. Another important reason for conducting the sensitivity analysis is that, in general, during early stages of software development, failure data are insufficient, which makes the exact values of parameters difficult to obtain. In the Condor case study, we focus on the sensitivity of two parameters: the component reliability and the transition probability between scenarios. In the sensitivity analysis of Condor we follow an approach similar to Lo et al. [LHC+05] for the sensitivity analysis, a small improvement compared to our previous work in in [RRU05b]. The only difference is that in the work of Lo et al. they conduct the sensitivity based on the *relative change* of the system reliability.

6.7.4.1 Component Sensitivity

We start from the analysis of the system's sensitivity to the component reliability. So let S_{ρ,R_i} be the sensitivity of the relative change of the system reliability to R_i, when R_i is changed by $100\rho\%$, that is:

$$S_{\rho,R_i} = \frac{\Delta R_s}{R_s} \tag{6.8}$$

where ΔR_s can be defined as the relative change of the system reliability, R_s, when component reliability R_i is changed by $100\rho\%$. In fact, we compute S_{ρ,R_i} as the normalization of the system reliability variation over the initial reliability (i.e. when the variation $\rho\%$ is zero). The ΔR_s is computed as follows:

$$\Delta R_s = R_s(R_1, \ldots, R_i + \rho R_i, \ldots, R_N) - R_s(R_1, \ldots, R_i, \ldots, R_N) \quad (6.9)$$

As a result of that computation, the sensitivity analysis concerning the relative change of the component reliability can be found and the most sensitive component identified, in a discrete manner.

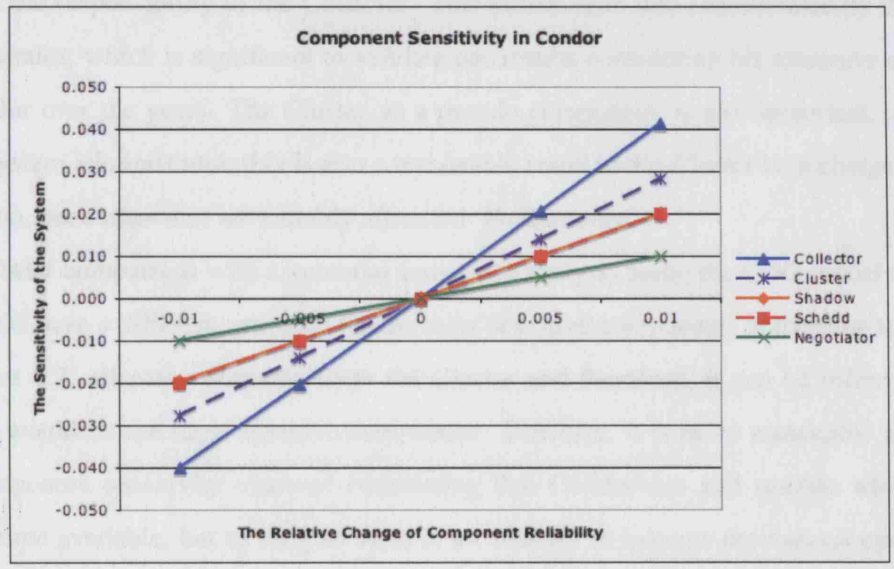


Figure 6.8: Analysis of the most sensitive component reliability in Condor

$\rho\%$	R_i	R_s	S_{ρ,R_i}
1%	98.5%	21.764%	0.041
0.5%	98.0%	21.328%	0.020
0%	97.5%	20.899%	0.000
-0.5%	97.0%	20.476%	-0.021
1%	96.5%	20.06.%	-0.040

Table 6.5: Values for the Collector Sensitivity

The methodology consists in obtaining the range of component reliability individually, annotating in LTSA with the modified values and then generating the new predictions. Finally, applying formula 6.9, we obtain the results presented in Figure 6.8. In Table 6.5 we present the values used to obtain the results depicted in Figure 6.8 for the Collector component, using various reliability values R_i, which are variations with the relative change $\rho\%$, with obtained system reliability R_s and sensitivity S_{ρ,R_i}.

The results show that Collector is the most sensitive component, followed by the Cluster, the Shadow (equally placed with the Schedd) and the Negotiator, respectively. From the Condor

scenarios, we can see that the Collector is indeed a very important component for the Condor system. The Collector processes two important messages: (1) the two *publish* messages, where available resources are published to the Central manager and the *obtain* in the process of allocating a resource to a job. Therefore, the intelligence of the matching between the job and the resource that will be allocated (or re-allocated after a failure in an allocated resource) to that job is all in the responsibility of the Collector. This information was corroborated by the Condor administrator, which is significant to validate our results considering his extensive experience in Condor over the years. The Cluster, as a pseudo-component, is also important. According to the system administrator, this is also a reasonable result as the Cluster is in charge of the job execution, once resources are suitably allocated. Furthermore,

A brief comparison with a potential sensitivity analysis using the CDG model shows that we would have a different outcome for the most sensitive component. According to the CDG in Figure 6.7, all paths originate from the Cluster and therefore, it can be inferred that the Cluster would be the most sensitive component. However, it is more reasonable to consider our component sensitivity outcome considering that Condor can still operate when there is no resource available, but as long as there is a Collector to manage the various operations in Condor.

6.7.4.2 Scenario Transition Sensitivity

The next sensitivity analysis we conduct in Condor is for the scenario transitions. In order to compute the scenario transition sensitivity, we follow an analogous methodology we applied for the component sensitivity, but varying the transition probabilities instead for each set of branching transitions. We compute the ΔPTS_{ij} as follows:

$$\Delta PTS_{ij} = R_s(PTS_{11}, \ldots, PTS_{ij}+\rho PTS_{ij}, \ldots, PTS_{NN}) - R_s(PTS_{11}, \ldots, PTS_{ij}, \ldots, PTS_{NN}) \tag{6.10}$$

Therefore, the sensitivity analysis concerning the relative change of the scenario transition probability can be found and the most sensitive transition identified, in a discrete manner.

There are two sets of transitions we analyse: (1) from scenario Obtain to scenarios Allocate and UsrCancel and (2) from scenario JobExecution to scenarios Shutdown and Cancel. The basis of the variation on set 1 was the transition from Obtain to Allocate, and the basis of the variation on set 2 was from JobExecution to Shutdown. We present the results for the Obtain–Allocate transition in Table 6.6 we used to obtain the curve depicted in and in Figure 6.9 for the Obtain–Allocate transition. The same procedure follows for the transition JobExecution–Shutdown. It is important to note that the variation of the values on the x axis of Figure 6.9 is based on the transitions in sets 1 and 2. Therefore, if the obtained curve has an increasing

or decreasing effect, it refers to those transitions. An analysis using Obtain–UsrCancel and of JobExecution–Cancel instead, would have the exact inverted behaviour.

$\rho\%$	PTS_ij	R_s	$S_{\rho,PTS_{ij}}$
2%	96%	20.69	-0.010
1%	95%	20.78	-0.006
0.5%	94.50%	20.84	-0.003
0%	94%	20.89	0.000
-0.5%	93.50%	20.96	0.003
-1%	93%	21.03	0.006
-2%	92%	21.19	0.014

Table 6.6: Values for the Allocate Transition Sensitivity

From the data obtained and depicted in Figure 6.9, it is evident that the transitions from scenario Obtain to scenarios Allocate and UsrCancel are more sensitive than the transitions from JobExecution to Cancel and Shutdown. It shows that from those transitions, the Obtain–UsrCancel is the most sensitive of them all, considering that when the transition to Allocate increases in probability, the system reliability decreases (or equivalently, when the transition to UsrCancel increases in probability, the system reliability also increases). This behaviour can be explained by the fact that the transition to UsrCancel leads to the shortest path to the End scenario, which is a reasonable conclusion as the shorter the path, the less processing of information there will be and therefore the less likely a fault will happen.

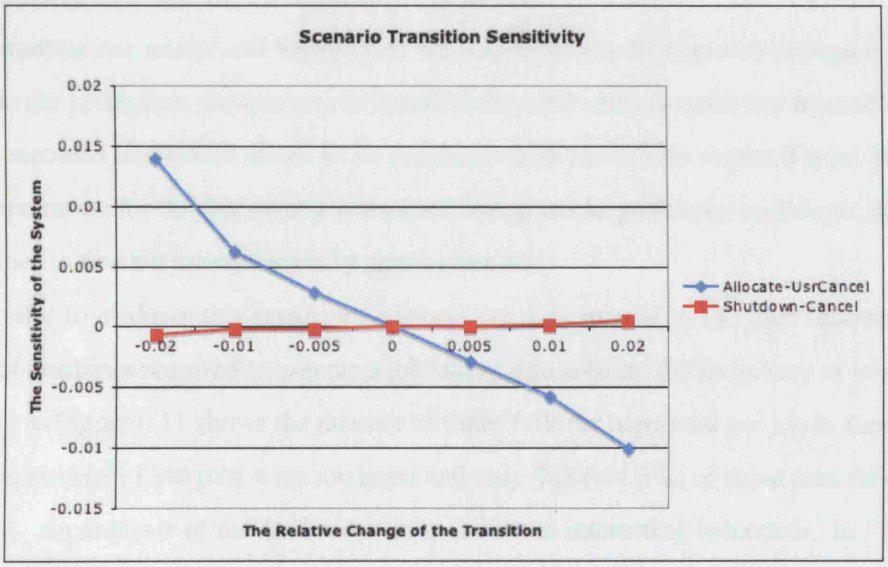


Figure 6.9: Analysis of The most sensitive scenario transition in Condor

6.7.5 The Cluster Size Analysis

Another application of our prediction technique in Condor is to determine how much replication is needed for different values of desired reliability, and how many independent jobs can be accommodated at a desired reliability through further replication. Condor is not an ideal case study since it achieves its reliability primarily through massive replication, which existing approaches cannot model explicitly (due to model complexity and state explosion). Nevertheless, the rationale for Condor's level of replication is not documented anywhere, so we can apply our approach to explain the effects of its level of replication. In this section we analyse how the number of nodes affects the reliability of Condor to execute a job from the perspective of our reliability analysis technique.

We also compute the effect of node replication on Condor's reliability through the approach of Wang et al. [WWC99], which extends the Cheung approach to estimate the reliability of heterogeneous software architectures, including fault-tolerant ones. We then compare both outcomes and extend our analysis to compare with the results obtained from logs of the failover behaviour of the jobs.

To conduct this analysis, we consider the node behaviour in the presence of a fault is to continually failover a job to another node as necessary until the job is completed. It is important to notice here that we do not intend to model and anlyse the parallelization of jobs in Condor. We are analysing here the effect of one job per resource at a time. We apply the canonical parallel configuration of component replicas as presented in Formula 6.2 to estimate the reliability of the cluster for a particular job and use our technique to estimate Condor's overall reliability for each cluster reliability estimation.

Computing our model and Wang et al. we obtain the results depicted in Figure 6.10. According to our prediction, the increase in the reliability of Condor is quite low beyond five nodes for each executed job, which seems to be consistent with the results obtained from Wang et al. Our interpretation for this estimation is that, according to our prediction technique, the number of machines to finish a job successfully approaches six.

In order to evaluate this result, we analyse the data related to the jobs failover, i.e. the number of machines required to execute a job taking into account the frequency of jobs failover. The graph in Figure 6.11 shows the number of times failover happened per job in June and July 2007. The total of 11340 jobs were analysed and only 482 ($\approx$4.3%) of those jobs failed over at least once. An analysis of the failover of jobs shows an interesting behaviour. In Figure 6.11 we depict the relationship between the number of jobs and the number of times they failed over until they finished their execution. A first glance at the discrete data seem to fit a hybrid of

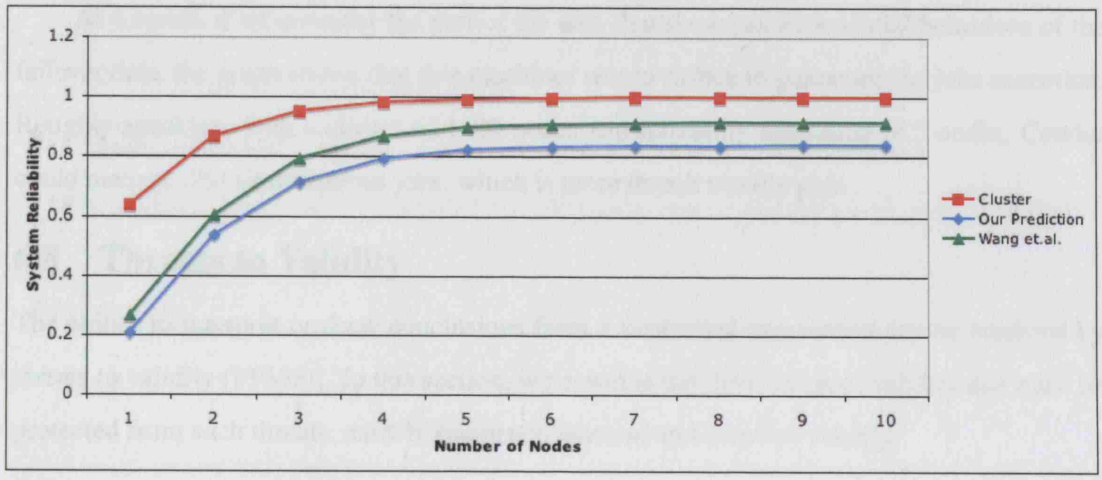


Figure 6.10: Predicted Impact of the Number of Nodes per Job

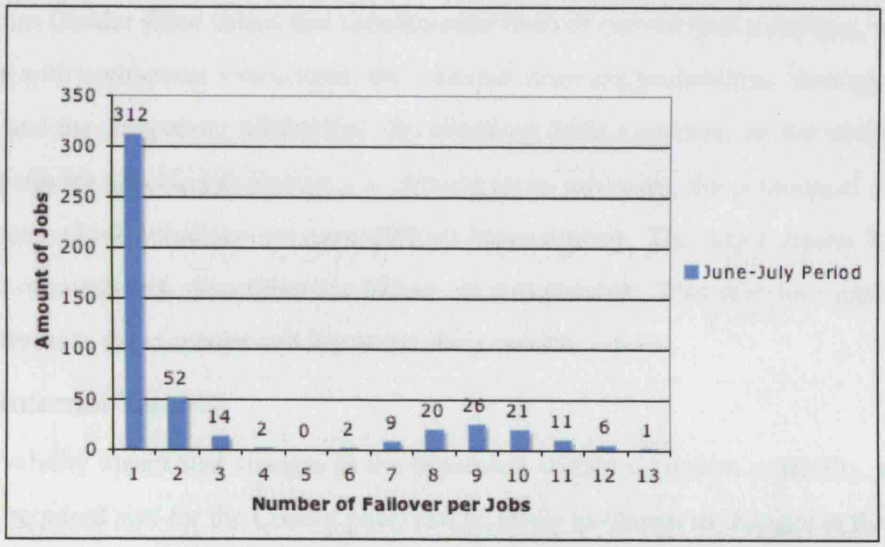


Figure 6.11: Failover of Jobs in the Period of June and July 2007

two distributions: one where it decreases exponentially until the number of failover per job reaches five and another one where it seems to fit into a normal distribution from six to thirteen failovers per job. However, when we analyse the failover data, there are only two batches of job (i.e. 326230 and 326231) which were responsible for the failover beyond 6 nodes. According to the Condor administrator, it is likely that those two batches were submitted by the same user and they were either experimental batches of job or there was an error in the specification of the job. Therefore, that 'tail' in bell shape should not be considered as a typical behaviour of job failover. The Condor administrator also agreed that no job should failover to more than four or five machines, from his past experience.

As a result, if we consider the part of the data that shows an exponential behaviour of the failover data, the graph shows that five machines would suffice to guarantee the jobs execution. Roughly speaking, with a cluster of 1400 nodes and reliability saturation of 5 nodes, Condor could manage 280 simultaneous jobs, which is more than it usually gets.

6.8 Threats to Validity

The ability to interpret or draw conclusions from a controlled experiment can be hindered by threats to validity [PPV00]. In this section, we describe the three kinds of validity that must be protected from such threats, namely *construct*, *internal* and *external* validity.

6.8.1 Construct Validity

Accomplishing this validity means that the independent and dependent variables accurately model the abstract hypothesis questions. In order to answer those questions, we first need to obtain from Condor those values that constitute the basis of our analysis technique, namely the scenarios with component interactions, the scenario transition probabilities through the usage profiles, and the component reliabilities. By obtaining those measures, we are able to test the three hypotheses specified in Section 6.1. Among those measures, the component reliabilities are the ones which constitute the most difficult measurement. The major reason for that the difficulty of accurately classifying the failures of components. This therefore represents the greatest threat to the construct validity of the study results.

6.8.2 Internal Validity

Internal validity means that changes in the dependent variables (system reliability, sensitivity analysis, required size for the Condor pool) can be safely attributed to changes in the independent variables (scenarios and their transition probabilities, component reliabilities, job submissions and their properties, the number of execution machines, the failover overhead of Condor for execution machines). In order to assess the internal validity of the variables, we consider the following aspects:

1. Collecting data extensive enough to provide reliable information.

2. Verifying the assumption of independence of failure of components. Expert knowledge from the Condor administrator already confirmed to us that all the components in Condor experience independence of failures among themselves. Therefore, once one of the component fails to start, it automatically tries to re-initiate, unless it is due to a failure in the server hosting the component.

3. Verifying if our scenario model is comprehensive enough to account for the various elements composing Condor and if those elements are sufficient to characterize Condor reliability.

4. Accounting for implied scenarios in Condor's predicted reliability, in case implied scenarios are detected.

We thus feel that our study has comprehensively covered all aspects of Condor that could influence the dependent variables of the study.

6.8.3 External Validity

External validity means that the case study's results generalize to settings outside the study. We have of course studied only a single system, so additional studies are needed to assess the generality of the results. But reliability analysis applies to the component-based development approach, where clear interfaces between the components, their message exchanges and their usage need to be obtained. What constitutes a major limitation of our technique is the feasibility of obtaining measures for component reliabilities that can be characterized in terms of successful message invocations.

As previously presented in Section 6.5, we managed to make an equivalence between the components availability and their successful message processing. As long as the class of failure is well distinguished and coherent throughout the measures, there should be no restriction in terms of categories of component reliability types for our reliability analysis technique.

Another important aspect to consider in terms of external validity of our system is that we do not take into account the time limit for a message to be executed. Therefore, our technique may not be suitable to those systems with long running method invocations , or to real-time systems whose correct functioning is dependent on the time at which the results are produced. On the other hand, if time is not an issue, in particular, if method invocation *times* are not a significant factor for the reliability of a system (as seems to be the case for many well-designed component-based and object-oriented systems), then our technique can be useful to analyse component-based systems with concurrent behaviour, where events can occur at irregular intervals. This is due to the fact that the component interactions are modelled as concurrent processes in LTSA, and the system model to be analysed from the reliability perspective is a composition of those processes.

6.9 Critical Evaluation

Qualitatively, using Condor to evaluate our reliability prediction technique confirmed some of the issues already known in the field of component-based software reliability and revealed some others. It is already known that estimating component reliability is one of the greatest challenges in the field of component-based software reliability [GPT01], firstly for the lack of a clear association between components and their failures, and secondly for the lack of a classification of those failures in log data .

Before the case study with Condor, we attempted to conduct a case study with an open source system called ActiveBPEL. Obtaining the independent variables for our analysis technique in ActiveBPEL posed one a major challenge, obtaining the failure data for the components. The progress of that case study was impeded for two reasons: first for the lack of clear information in the bug track reports to relate failures with their components, and second a classification of severity of the failures. That information required expert knowledge of the system that could not be easily obtained, in spite of the various attempts we made . In order to avoid running the risk of not obtaining usable data in a manageable amount of time, we had to halt our efforts on that a case study and find a system where that information was at hand.

Fortunately, those problems were overcome in Condor for most of the cases as to obtaining Condor component failure data, except for the hundreds or thousands of Starter and Startd, encapsulated as the pseudo-component Cluster, running on the Condor pool of machines. As presented in previous sections, we managed to turn around the problem of estimating the reliability of the Execution machine components. As the case study progressed, we realized that some of the analysis results, especially those related to measuring Condor reliability, were hindered by the various fault tolerance features embedded in the system. Those features hid failures arising from the system which rendered infeasible obtaining an actual reliability estimate for the one-node cluster configuration of the system that we were able to model.

One of our initial concerns before starting the case study with Condor was whether we would be able to model it in LTSA and make a satisfactory abstraction of the model, enough to synthesize an architecture with a reasonable set of states, without state explosion. Thanks to the expert support of the Condor administrator, an abstract model of Condor with a relatively small set of states was developed. As a result, the facility of using scenario specifications for reliability modelling showed itself very efficient on weaving the stakeholder's architecture view of the system with the reliability model. Therefore, we find this closeness, regarding the granularity of the system states, an advantage to the stakeholder compared to previous architecture-based reliability models.

Following the discussion about comparison with other reliability analysis techniques, which is initially our hypothesis one, we realized that within some reasonable range our technique is close to most of the existing architecture-based reliability models, in spite of the infeasibility of comparing with the actual reliability of the system. On the other hand, previous work in the literature [GPHP05], particularly where composite and hierarchical architecture-based models are related, has evaluated those other models. That gives some degree of confidence that our technique, though conservative, is following a sound approach. However, Vittorio et al. in [CG07] point out to the discrepancy that can occur when error propagation is not taken into account, particularly in architecture-based approaches. The pessimistic outcome of the reliability predictions presented in Table 6.4 may reflect that fact. Therefore, the Cheung method we use may need to be improved to generate more accurate results by considering the factors related to error propagation, as shown by Vittorio et al.

We also managed to accomplish the sensitivity analysis for the components and for the transitions that have more impact on the reliability of Condor, as stated in our hypothesis two. The Collector and the Cluster were those which had the greatest impact on the system reliability of Condor. The first was reasoned by the fact that Collector is most involved in processing information about the resources to be allocated and the Cluster to be most involved in the execution of the job, once a resource was allocated. As for the transition sensitivity, the Obtain–UsrCancel demonstrated the higher impact on the reliability outcome, and again this was justified by an understanding of the system behaviour.

As for hypothesis three, we realized a major impact on the performance of the model computation when we removed the potential deadlock scenario, which was the consequence of an implied trace not specified in the first scenario specification. As for the improvement on the quantification of the predicted reliability, this could not be verified. If we compute the reliability of Condor without taking the implied scenario into account, we obtain 20.92% for the reliability of Condor (0.03% difference). Therefore, considering this minor difference, we cannot conclude from our case study that there is indeed an improvement on the quantification of the predicted reliability once implied scenarios are taken into account. However, the model was definitely more manageable to be computed in LTSA (from several to few minutes) once the scenarios were enhanced.

Finally, from the resource planning perspective, our reliability analysis technique also provided information consistent with observed data and comparative techniques. However, we recognize the need to extend our study to a model in which multiple jobs are concurrently executed following the robust parallel infrastructure of Condor.

Chapter 7

Conclusions and Future Work

The main purpose of this thesis has been to come up with a technique to support reliability engineering in a model-driven software engineering process. In particular, we provided means to realize that technique in the very early stages of the software development process based on annotated scenario specifications taking into account the concurrent behaviour of component-based software systems. Those scenarios are then translated into a fine-grained probabilistic LTS, which is then interpreted according to a DTMC model. Through the technique we provide a bird's eye view of the system with focus on reliability while also validating the scenario specifications according to the intended behaviour of the software by means of implied scenarios analysis. A plugin for reliability analysis was produced to support the technique, which was also evaluated on a real life case study. Additionally, we defined a standard way to represent the reliability engineering process in the OMG's Model Driven Architecture, which defines basic mechanisms to structure models consistently and formally express the semantics of models in a standard way. We propose through our UML profile for software reliability an amendment to the standard OMG profile for Quality of Service and Fault Tolerance so that the reliability engineering process can be seamlessly integrated into the standard software engineering development process. In this chapter we restate our contributions, critically evaluate the attained results and highlight directions for future work.

7.1 Conclusion

The purpose of our work in this thesis is to concentrate on the reliability assurance of the system from requirements to deployment, with adequate analysis, integrated through mapping rules. Concentrating on these levels, we believe that the desired reliability of software systems is perceived in later software development phases according to the required reliability property defined at the architecture level. All in all, there are two major contributions in our thesis.

Reliability Prediction by Model Synthesis From Scenarios

This contribution consists of a method to predict software system reliability from scenario specifications by extending a scenario specification to model (1) *the probability of component failure*, and (2) *scenario transition probabilities* derived from an operational profile of the system. The contribution of this approach is a reliability prediction technique that takes into account the component structure exhibited in the scenarios and the concurrent nature of component-based systems. The analysis also comprises sensitivity analysis to identify components and usage profiles with greater impact on the system reliability. Through the sensitivity analysis we can find out how the system reliability is sensitive to the (1) *component reliabilities* and (2) *scenario transition probabilities*. These two analyses can help us to identify components and scenario transitions that could represent a threat to the reliability of the software system. Taking into account the concurrent nature of component-based software systems, we are also able to analyse what effects the prevention of undesirable implied scenarios cause to our reliability prediction technique.

We evaluated our reliability analysis technique through a significant case study with Condor, a distributed job scheduler and resource management system. We focus on evaluating the technique aiming at quantitatively comparing our technique with other sound reliability prediction techniques as well as determining the suitability of using our technique to analyse a system in the magnitude of Condor. We compared our results in Condor with major architecture-based software reliability analysis techniques. Results were within range of most techniques where ours can be considered the most conservative ones. The models having results with great discrepancies were mostly due to deficiencies identified on those models. Finally, we also managed to accomplish sensitivity analysis for the components and for the transitions that have the greatest impact on the reliability of Condor.

Conforming Reliability Modeling to a Standard

In this contribution we provide a solution for model-driven reliability engineering following the principles of the MDA standard. This contribution focuses particularly on addressing reliability

modeling and analysis into MDA. The syntax and semantics of MDA models are represented through profiles that extend the core UML using metamodeling techniques. By this means, modeling reliability is considerably facilitated as the complexity of making software more reliable is raised to a higher level of abstraction. To achieve this goal, this contribution relies on reference models specifications such as [OMG05] as well as extensions of the UML metamodels. The previous UML profile for Quality of Service and Fault Tolerance did not address the modeling of dynamic aspects (such as scenarios, component interactions, and operational profiles) often required in modeling component-based software reliability. We propose to amend that profile with those features and thus to semantically integrate analysis and design models into one environment. The result of our proposal is a comprehensive framework for software reliability modeling, from elicitation of system requirements to design of reliability mechanisms to analysis and evolution of software reliability all following standard model driven engineering of software systems.

By means of a reliability profile, the architecture of an application can express both method invocations and deployment relationships between the application components. On the whole, the reliability profile comprises three other major profiles: the design, the analysis and the deployment profiles. In the design profile, meta-modeling techniques are used to map the reliability property and fault tolerance mechanism into a profile. As replication is a common practice to achieve fault tolerance, we choose to represent it as a fault-tolerance mechanism comprised in the reliability profile we propose. In the analysis profile, the reliability property for the system is verified and a set of rules for the mapping between design and analysis is specified. Finally, in the deployment profile, components are modeled in a distributed configuration according to the target reliability for the system. These three profiles mainly extend two specifications: (1) the UML Profile for Schedulability, Performance and Real-Time Specification (SPT Profile) [OMG05] and (2) the UML Specification [OMG04a].

At a platform-specific level, where we define a PSM for reliability, we target the EJB for being a mature component-based middleware and which offers very modular and simple ways to implement fault-tolerance features, compared to the Fault Tolerant CORBA architecture. We apply our PSM on an well known EJB vendor called JBoss, where we make a parallel between the profile elements and rules in the EJB PSM with JBoss fault tolerant elements for EJB. As for the profile for reliability analysis, we believe it is natural so that the approach of using scenarios for reliability analysis is (or has even greater potential to become) widely adopted by software engineers. As a result, that profile is meant to enhance the current QoSFT profile, so that other reliability analysis techniques can benefit from the standard. We show through our motivating

example how the profile can be applied from the UML environment to the formal analysis of LTSA and back again to enhance the UML model.

7.2 Future Work

Our purpose to realize a comprehensive methodology for component-based reliability starting from early stages of software development showed us there are still some directions to follow for future work.

Include Other Probabilistic Models Into the Reliability Prediction Analysis

There are other mature and sound model checking tools such as PRISM [KNP04], which support the use of various probabilistic models. However, those tools do not benefit yet from the various advantages of model synthesis from scenario specifications. We have used in our prediction technique the simplest probabilistic model, which is the DTMC, as a variant of Markov Chain. A DTMC is fully probabilistic, which gives rise to the fact that the parallel composition of the components (or processes to be more consistent with FSP syntax) follows a generative semantics, where probabilities are distributed over all outgoing transitions from a state [DHK00].

However, there are two other probabilistic models systems may follow. For example, some systems may contain a non-deterministic behaviour and therefore require a probabilistic parallel composition to suit that situation. In this case, another probabilistic model is required, such as the Markov decision process (MDPs) as those used in PRISM [KNP04]. In that case, instead of following a deterministic behaviour, the environment chooses the next state only after receiving an input by following a probability distribution assigned to the input, i.e. a *reactive* system type of parallel composition [SV04] As a result no action can be taken from a state in a automata (or a pLTS) unless it's probability function defined.

Finally, there is another probabilistic model which allows the modelling of applications that run continuously. Those models are called Continuos-time Markov Chains (CTMC) and can combine properties of generative and reactive probabilistic automata. In that case, *for every input action there is a reactive transition*, while *at most one generative probabilistic transition gives the output behaviour of each state* [SV04]. This probabilistic model follows the I/O system type of parallel composition.

In practice, choosing a probabilistic model and parallel composition semantics for the scenario specification impacts on how the transitions are going to be annotated and how the architecture model LTS composed of the various component interactions is going to be synthesized in LTSA. As a result, being able to accommodate those probabilistic models means we extend the capability of modelling a broader range of systems.

Include Time as a Property of Reliability

Standard definition of reliability states it as "the probability of failure-free software operation for a specified period of time in a specified environment" following Lyu's Handbook on Software Reliability Engineering [Lyu96], or as"the ability of a system or component to perform its required functions under stated conditions for a specified period of time" following the IEEE Standard Computer Dictionary' [IEE90].

Through our reliability prediction technique we were interested in computing, in general lines, as a function of the state reachability of the systems, following Cheung's approach. Therefore, it does not allow one to make any analysis for real-time applications. In that case, we need to extend the model to allow the inclusion of time as a property into the specification following other mature approaches in the field, such as those used in PRISM [KNP04]. In other to do that, we need to extend scenario specifications with real-valued time, costs and rewards.

Analyse for Implied Scenarios as a Property in Reliability Analysis

One of the benefits of model synthesis through scenario specifications is the analysis for inconsistencies between the user specified models and the synthesized ones, i.e. the implied scenarios. We have pointed out in our thesis the impact of implied scenarios in the reliability analysis. However, we recognize that there might be situations where those implied scenarios, especially when they are negative, they may even decrease the software reliability from a quantitative point of view, although from a qualitative aspect it is an improvement. Or suppose the nature of the implied scenario raises a special kind of software failure such as a byzantine failure and we want to analyse the probability that particular scenario is executed.

In those cases, we may use approaches similiar to the work of Giannakopoulou and Magee [GM03], where *fluent* propositions are introduced to express properties that combine state and action. Fluent propositions define state predicates whose values are determined by the occurrence of actions. More specifically, a initial step could be to model the final state of a detected negative implied scenario as a final error transition. As a result, the model checking for that implied scenario and the computation of its probability of occurrence is simply reduced to a reachability search for the ERROR absorbing state.

Extend the MDA Profiles for Other Fault Tolerance Mechanisms

Finally, another direction for future work is to complete the models for fault tolerance support in the MDA. We worked with replication as the major fault tolerance mechanism. Other features to explore are transaction management and the messaging service. The first feature has been initially tackled by Loecher [Loe04], where he presents a conceptual framework for model-based

declarative transaction service configuration. The author recognizes the limitations of a declarative approach, but he targets model-based configuration to address those potential problems.

The messaging service, on the other hand, still seems to be uncovered. Messaging services allow loosely coupled, reliable, asynchronous communications between components in a distributed fashion. From the perspective of current distributed component-based architectures, messaging has become more and more popular due to the facilities it promotes. Therefore, it seems a natural step for us to accommodate that kind of service into standard model-driven development.

Appendix A

The OCL Scripts

Before we specify the OCL implementation of the rules we use some other operations in OCL for convenience. The operations are:

We import the operation *isStereotyped* from the UML Profile for CORBA Specification [OMG02a], which we transpose to this document for convience:

```
context: ModelElement
isStereotyped : (stereotypeName : String) : Boolean;
self.stereotype.name = stereotypeName
```

We also import the operation *allEnds* from [OMG02a], which results in a Set containing all AssociationEnds for which the Classifier is the type:

```
context: Classifier
allEnds : Set(AssociationEnd);
allEnds = self.associations->collect(assoc | assoc.connection)
```

Finally, we transpose here the operation *allConnections* from UML 1.5 Specification [OMG03a], which results in the set of all AssociationEnds of the AssociationClass including all connections defined by its parent.

```
context: AssociationClass
allConnections : Set(AssociationEnd);
allConnections = self.connection->union(self.parent -> select
 (s | s.oclIsKindOf(Association)) -> collect (a: Association |
    a.allConnections)) -> asSet
```

We now define the rules in OCL for the MDA profile for reliability engineering in the following sections.

A.1 The Reliability Analysis UML Profile Rules

The major modelling elements of the reliability analysis profile we built is centered on the HMSC and the BMSC scenarios, which are addressed in UML 2 [OMG04a] as Interaction Overview Diagram and Sequence Diagrams respectively. However, a detailed abstract syntax for the Interaction Overview Diagram is not provided in the UML specification, which makes unclear for some model elements the way the navigation through OCL constraints should be specified. The best definition for the Interaction Overview Diagram is on page 499 of the referred specification, which states the following:

> Interaction Overview are specialization of Activity Diagram that represent Interactions. Object Nodes are replaced by Interactions or InteractionUses. (...) In place of ObjectNodes of Activity Diagrams, Interaction Overview Diagrams can only have either Interactions or InteractionUses. (...) Decision Node and a corresponding Merge Node represent Alternative Combined Fragment ...

Based on this definition, we devise the following OCL constraints for the UML profile for reliability analysis:

Rule 1 – Within an *Interaction Overview Diagram* stereotyped as an *HMSC*, every node must be either a *BMSC* or another *HMSC*. In order to define this rule, we follow the definition of Interaction Overview Diagram according to the UML 2 Specification: "Interaction Overview Diagrams are specialization of Activity Diagrams that represent Interactions.

```
context Interaction Overview
inv: self.isStereotyped("HMSC") implies
       self.ownedElements -> forAll (h | h.isTypeOf(Interaction)
        implies h.isStereotyped("HMSC")
   or h.isStereotyped("BMSC"))
```

Rule 2 – Every *HMSC* and *BMSC* must be uniquely named.

```
context Interaction Overview
inv:
self.isStereotyped("HMSC") implies
  self.ownedElements -> forAll(h | h.isStereotyped("HMSC")
     implies self.ownedElements -> isUnique(h.name)
          and h.name <> self.name)
 inv:
  self.isStereotyped("HMSC") implies
  self.ownedElements -> forAll(b | b.isStereotyped("BMSC")
```

– The reason behind the line below is that a node, as an Interaction may have to be repeated in case it derives from a Decision Node with a PTS tag value.

```
and not h.edge -> exists (e | e.isKindOf(DecisionNode)
implies self.ownedElements -> isUnique(b.name))
```

Rule 3 – Every *HMSC* must have one *Activity initial node* and one *Activity final node* (stereotyped as *Stop*) only.

```
inv:
context Interaction Overview
inv: self.isStereotyped("HMSC") implies
       (self.ownedElements -> select (h | h.isTypeOf(ActivityFinal)
            and h.isStereotyped("STOP")) -> size() = 1)
            or (self.ownedElements ->
               select (h | h.isTypeOf(InitialNode)) -> size() =1)
```

Rule 4 – *HMSC* nodes must have at least one incoming and one outgoing transition, except the *initial node* and *final node*.

```
context Interaction Overview
inv:
      self.isStereotyped("HMSC") implies
      self.ownedElements ->
      forAll(n | n.oclIsKindOf(ActivityNode) implies
             n.incoming->size() >= 1 and
             n.outgoing->size()>=1 and
             not n.oclIsKindOf(ControlNode))
```

Rule 5 – The *PTS* values of *HTransition*-stereotyped nodes connected to the same *Decision* node within an *HMSC* must sum to one.

```
context ControlNode
inv:
 self.isTypeOf(DecisionNode) implies
   self.outgoing->collect(a| a.isStereotyped("HTransition") and
      a.taggedValue->select(type.name="PTS")) -> sum() =1
```

A.2 The Rules to Map from Design to Deployment

The following rules apply for the mapping to the deployment model

Rule 1 – For a replica to be clustered, the `Replica` stereotype must have a non-null `Clustered` tag, a boolean value.

```
context: Classifier
pre: self.isStereotyped("Replica")
        implies self.taggedValue->exists(m | m.type.name =
                    "Clustered" and m.dataValue.notEmpty())
```

Rule 2 – For each deployed replica of the same component, the node name and the host address should be unique.

```
context: Component
inv:
self.client->forAll( m1 |
       self.client -> forAll (m2 |
           m1.oclAsType(Node) and  m1.isStereotyped("Replica") and
           m2.oclAsType(Node) and  m2.isStereotyped("Replica") and
  (m1.name = m2.name or m1.host = m2.host) implies m1 = m2))
```

Rule 3 A <relDeploys> stereotyped Association must have the Classifiers of its Association Ends stereotyped as a <Replica>.

```
context: Association
inv:  self.isStereotyped("relDeploys")
        implies self.connection ->
            forAll(a | a.participant implies a.isStereotyped("Replica")
```

Rule 4 – Define what the offered and the required reliability assurance of each component is, according to what is provided in the <<relRequires>> realization mapping. By this means, it is possible to quantify the number of replicas each components should have in the cluster to guarantee their required reliability level.

The OCL rule to verify the number of replicas in the deployment satisfy the required reliability of each component is defined in OCL as follows:

```
package Foundation::Core

context Abstraction def:
```

- *Calculating the number of replicas* n

```
  let exp(x : Real, y : Real) : Real =
     if(y = 1) then x
  else x * self.exp(x, y - 1)
     endif
```

- Identifying the Mapping Profile

```
inv:
  self.stereotype->forAll(name = "mapping") implies
```

- Reading from every EJB component stereotyped as "Replica"" the tagged value "OfferedValue"", which is the actual reliability property of the Enterprise Bean component, and the tagged value "RequiredValue"", which is the reliability to be achieved

```
  self.client.oclAsType(Classifier).ownedElement->forAll(
      m : ModelElement | m.oclIsTypeOf(Instance) implies
         (1 - self.exp((1 - m.taggedValue->any(type =
              "OfferedValue").dataValue),
        self.supplier.oclAsType(Classifier).ownedElement->select(
               n : ModelElement | n.oclIsTypeOf(Instance) and
      m.stereotype->exists(name = "Replica") and
               n.name = m.name)->size())) >
           m.taggedValue->any(type =
              "RequiredValue").dataValue)
```

A.3 The EJB Transformation Rules for Deployment

As we mentioned in Chapter 5, the configuration of EJB for replication is mostly done through deployment descriptor files, following the principles of modularization of EJB. The class containing those configuration parameters are under the metaclass stereotyped as `relDeploymentSpec`. The rules to realize the EJB configuration are then defined as follows:

Rule 1 The `ReplicaSession` or the `ReplicaEntity` inherent the attribute `ReplicaID` from the parent `Replica`. The `ReplicaID` corresponds to the bean name and in JBoss, the bean name is specified in the `<ejb-name>`the `ReplicaID` is mapped to the Bean name itself, `<ejb-name>` attribute in JBoss.xml file. We define this rule under the metaclass stereotyped `relDeploymentSpec`.

```
context AssociationClass
inv:  self.isStereotyped("relDeploymentSpec")
        implies self.allConnections()->
             exist(c | c.participant.isKindOf(Component) and
                 c.isStereotyped("Replica") and
                 c.taggedValue-> exists (m | m.type.name =
                  "ReplicaID" and m.dataValue = self.ejb-name)
```

Rule 2 The `EJBGroupManager` (which maps to the properties similar to `<cluster-config>` element in the JBoss.xml descriptor file) inherents from the `GroupManager`

two major attributes: (1) the `EJBManager` identifier, inherited from `Manager` identifier (`ManagerID`), as the name of the partition (`<partition-name>` attribute in `<cluster-config>`), (2) the policy from the relationship with `PolicyManager`, which can be the first available node or round robin, applied to `<home-load-balance-policy>` and `<bean-load-balance-policy>` in `<cluster-config>`.

```
context AssociationClass
inv:  self.isStereotyped("relDeploymentSpec")
        implies self.allConnections()->
              exist(c | c.isStereotyped("EJBGroupManager") and
              c.taggedValue-> exists(m | m.type.name = "ManagerID" and
                              m.dataValue = self.partition-name)
inv:  self.isStereotyped("relDeploymentSpec")
        implies self.allConnections()->
             exist(c | c.isStereotyped("EJBGroupManager") and
             c.taggedValue-> exists(m | m.type.name = "LoadBalancePolicy"
                  and m.dataValue = self.load-balance)
```

Rule 3 If `TransientStyle` of the `ReplicaSession` class element, is `Stateful`, the descriptor requires a JNDI name, `<session-state-manager-jndi-name>`, initialized as Default.

```
context AssociationClass
inv:  self.isStereotyped("relDeploymentSpec")
        implies self.allConnections()->
             exist(c | c.isStereotyped("ReplicaSession") and
             c.taggedValue-> forAll(m | m.type.name = "TransientStyle"
                  and m.dataValue = "Stateful" implies
                          self.Jndi_name.notEmpty())
```

Appendix B

The XSLT script to translate MagicDraw UML Script into LTSA MSC Scenarios

This xslt script was created to convert UML diagrams designed in MagicDraw 9.5 into LTSA scenario specifications. Note however, that because Magic Draw did not implement the UML 2 Interaction Overview Diagram at the time we devised and implemented this translation, we decided to make use of Use Case diagrams instead. Therefore, the idea of flow between Use Cases had to be introduced in order to overcome that deficiency in that version of Magic Draw.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:
    UML="omg.org/UML/1.4" exclude-result-prefixes="UML">

<xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>

<xsl:template match="XMI[@xmi.version='1.2']">
  <specification>
    <xsl:apply-templates select="XMI.content/UML:Model/UML:Namespace.ownedElement/UML:
        Package/UML:Namespace.ownedElement" mode="HMSC"/>
    <xsl:apply-templates select="XMI.content/UML:Model/UML:Namespace.ownedElement/UML:
        Package/UML:Namespace.ownedElement" mode="BMSC"/>
  </specification>
</xsl:template>

<xsl:template match="UML:Namespace.ownedElement" mode="HMSC">
  <hmsc>
    <xsl:apply-templates select="UML:UseCase" mode="nodes">
      <xsl:with-param name="xpos" select="170"/>
    </xsl:apply-templates>
    <xsl:apply-templates select="UML:Dependency" mode="transition"/>
  </hmsc>
</xsl:template>

<xsl:template match="UML:Namespace.ownedElement" mode="BMSC">
  <xsl:apply-templates select="UML:UseCase" mode="bmscSpec"/>
```

```
</xsl:template>

<xsl:template match="UML:UseCase" mode="bmscSpec">
  <bmsc>
    <xsl:attribute name="name">
      <xsl:value-of select="@name"/>
    </xsl:attribute>
    <xsl:apply-templates select="UML:Namespace.ownedElement/UML:Collaboration/UML:
        Namespace.ownedElement" mode="ComponentMessages"/>
  </bmsc>
</xsl:template>

<xsl:template match="UML:Collaboration">
  <xsl:apply-templates select="UML:Namespace.ownedElement" mode="ComponentMessages"/>
</xsl:template>

<xsl:template match="checkIfSD">
  <xsl:param name="ok"/>
  <xsl:if test="$ok = isSD">
    <xsl:value-of select="isSD"/>
  </xsl:if>
</xsl:template>

<xsl:template match="UML:Namespace.ownedElement" mode="ComponentMessages">
  <xsl:for-each select="UML:ClassifierRole">
  <instance>
    <xsl:variable name = "xmiId" select="@xmi.id"/>
    <xsl:attribute name="name">
      <xsl:value-of select="@name"/>
    </xsl:attribute>
    <xsl:apply-templates select= "../../UML:Collaboration.interaction/UML:Interaction/
        UML:Interaction.message/UML:Message" mode="Messages">
      <xsl:with-param name="_instanceId" select="@xmi.id"/>
    </xsl:apply-templates>
  </instance>
  </xsl:for-each>
</xsl:template>

<xsl:template match="UML:Message" mode="Messages">
<xsl:param name="_instanceId"/>
  <!--  if exchanging messages do the following: -->
  <!-- if receiving message -->
  <xsl:if test="$_instanceId=@receiver">
    <input>
      <xsl:attribute name="timeindex">
        <xsl:value-of select= "XMI.extension/number/@xmi.value"/>
      </xsl:attribute>
      <name><xsl:value-of select="@name"/></name>
```

```
      <from>
        <xsl:apply-templates select= "../../../../UML:Namespace.ownedElement/UML:
            ClassifierRole" mode="id2name">
          <xsl:with-param name="id" select="@sender"/>
        </xsl:apply-templates>
      </from>
    </input>
  </xsl:if>
  <xsl:if test="$_instanceId=@sender">
    <output>
      <xsl:attribute name="timeindex">
        <xsl:value-of select= "XMI.extension/number/@xmi.value"/>
      </xsl:attribute>
      <name><xsl:value-of select="@name"/></name>
      <to>
        <xsl:apply-templates select= "../../../../UML:Namespace.ownedElement/UML:
            ClassifierRole" mode="id2name">
          <xsl:with-param name="id" select="@receiver"/>
        </xsl:apply-templates>
      </to>
    </output>
  </xsl:if>
</xsl:template>

<xsl:template match="mdElement" mode="SD">
<xsl:param name="id"/>
  <xsl:if test="@xmi.id=$id and umlType='Sequence Diagram'">
    <xsl:value-of select="@name"/>
  </xsl:if>
</xsl:template>

<xsl:template match="UML:UseCase" mode="nodes">
<xsl:param name="xpos"/>
  <bmsc name="{@name}" x="{$xpos}" y="130" />
</xsl:template>

<xsl:template match="UML:Dependency" mode="transition">
  <transition>
    <from>
      <xsl:apply-templates select="../UML:UseCase" mode="id2name">
        <xsl:with-param name="id" select="@client"/>
      </xsl:apply-templates>
    </from>
    <to>
      <xsl:apply-templates select= "../UML:UseCase" mode="id2name">
        <xsl:with-param name="id" select="@supplier"/>
      </xsl:apply-templates>
    </to>
```

```
  </transition>
</xsl:template>

    <xsl:template match="UML:UseCase|UML:ClassifierRole" mode="id2name">
     <xsl:param name="id"/>
        <xsl:if test="@xmi.id=$id">
          <xsl:value-of select="@name"/>
        </xsl:if>
    </xsl:template>
</xsl:stylesheet>
```

Bibliography

[A. 79] A. Goel and K. Okumoto. Time-Dependent Error-Detection Rate Model for Software Reliability and other Performance Measures. *IEEE Transactions on Reliability*, 28(3):206–211, 1979.

[AEY00] R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. In *Proc. of the* 22^{nd} *ICSE*, pages 304–313. ACM Press, 2000.

[AFMB03] T. Ayles, A. J. Field, Jeff Magee, and A. Bennett. Adding Performance Evaluation to the LTSA Tool. In *Tool demonstration, 13th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools*, September 2003.

[AK76] T. Anderson and R. Kerr. Recovery blocks in action: A system supporting high reliability. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 447–457, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[AL86] A. Avižienis and J.-C. Laprie. Dependable Computing: from Concept to Design Diversity. In *Proceedings of the lEEE*, volume 75(5), pages 629–638. IEEE Computer Society Press, 1986.

[ALR01] A. Avižienis, J. Laprie, and B. Randell. Fundamental Concepts of Dependability. In *IARP/IEEE-RAS Workshop on Robot Dependability*, May 2001.

[B. 06] B. Ban. Reliable Multicasting with the JGroups Toolkit. Technical report, Department of Computer Science - Cornell University, http://www.jgroups.org/javagroupsnew/docs/index.html, 2006.

[BGJP05] S. Bodoff, D. Green, E. Jendrock, and M. Pawlan. The j2ee^{TM} 1.4 tutorial. Technical report, Sun Microsystems, http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Ebank.html, December 2005.

[BL75] J. R. Brown and M. Lipow. Testing for software reliability. In *Proceedings of the international conference on Reliable software*, pages 518–527. ACM Press, 1975.

[BMM99] A. Bondavalli, I. Majzik, and I. Mura. Automatic Dependability Analysis for Supporting Design Decisions in UML. In Raymond Paul and Catherine Meadows, editors, *Proc. of the* 4^{th} *IEEE International Symposium on High Assurance Systems Engineering*. IEEE, 1999.

[Bug07] Bugzilla Team, The. Bugzilla. Technical report, http://www.bugzilla.org/docs/tip/html/, 2007.

[CEM04] R. Chatley, S. Eisenbach, and J. Magee. MagicBeans: a Platform for Deploying Plugin Components. In Wolfgang Emmerich and Alexander L. Wolf, editors, *Component Deployment*, volume 3083 of *Lecture Notes in Computer Science*, pages 97–112. Springer, 2004.

[CG07] V. Cortellessa and V. Grassi. A modeling approach to analyze the impact of error propagation on reliability of component-based systems. In Heinz Schmidt, Ivica Crnkovic, George Heineman, and Judith Stafford, editors, *Proceedings of the* 10^{th} *International Symposium, CBSE 2007*, volume 4608 of *Lecture Notes in Computer Science*, pages 140–156. Springer, 2007.

[Che80] R. C. Cheung. A User-Oriented Software Reliability Model. In *IEEE Transactions on Software Engineering*, volume 6(2), pages 118–125. IEEE, March 1980.

[CMI07] V. Cortellessa, A. Di Marco, and P. Inverardi. Integrating performance and reliability analysis in a non-functional mda framework. In Matthew B. Dwyer and Antonia Lopes, editors, *Proceedings of the* 10^{th} *Fundamental Approaches to Software Engineering*, volume 4422 of *Lecture Notes in Computer Science*, pages 57–71. Springer, 2007.

[CP04] V. Cortellessa and A. Pompei. Towards a UML profile for QoS: a contribution in the reliability domain. In *Proc. of the* 4^{th} *WOSP*, pages 197–206. ACM Press, 2004.

[CSC02] V. Cortellessa, H. Singh, and B. Cukic. Early reliability assessment of uml based software models. In *Proceedings of the* 3^{rd} *WOSP*, pages 302–309. ACM Press, 2002.

[CWT+04] C. Chapman, P. Wilson, T. Tannenbaum, M. Farrellee, M. Livny, J. Brodholt, and W. Emmerich. Condor services for the global grid: Interoperability between condor and ogsa. In *Proceedings of the UK E-Science All Hands Meeting, Nottingham*, 2004.

[DHK00] P. R. D'Argenio, H. H., and J-P. Katoen. On generative parallel composition. In Christel Baier, Michael Huth, Marta Kwiatkowska, and Mark Ryan, editors, *Electronic Notes in Theoretical Computer Science*, volume 22. Elsevier, 2000.

[DTL02] D.Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., December 2002.

[DXP06] Y. S. Dai, M. Xie, and K. L. Poh. Reliability of grid service systems. *Comput. Ind. Eng.*, 50(1):130–147, 2006.

[Emm00a] W. Emmerich. *Engineering Distributed Objects*. John Wiley & Sons, Inc, 2000.

[Emm00b] W. Emmerich. Software Engineering and Middleware: A Roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*, pages 119–129. ACM Press, April 2000.

[Emm02] W. Emmerich. Distributed Component Technologies and Their Software Engineering Implications. In *Proc. of the* 24^{th} *ICSE, Orlando, Florida*, pages 537–546. ACM Press, May 2002.

[FHLS98] P. Frankl, R. Hamlet, B. Littlewood, and L. Strigini. Evaluating testing methods by delivered reliability. *IEEE Transactions on Software Engineering*, 24(8):586–601, 1998.

[GLT98] S. Gokhale, M. Lyu, and K. Trivedi. Reliability Simulation of Component Based Software Systems. In *Reliability Simulation of Component Based Sofware Systems*, pages 192–201. Proc. of the 9^{th} ISSRE, 1998.

[GM03] D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. *SIGSOFT Softw. Eng. Notes*, 28(5):257–266, 2003.

[GP03] G. Ping Gu and D. C. Petriu. Early Evaluation of Software Performance Based on the UML Performance Profile. In *Proc. of the 2003 CASCON*, pages 66–79. IBM Press, 2003.

[GPHP05] K. Goševa-Popstojanova, M. Hamill, and R. Perugupalli. Large empirical case study of architecture-based software reliability. In *ISSRE '05: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, pages 43–52, Washington, DC, USA, 2005. IEEE Computer Society.

[GPMT01] K. Goševa-Popstojanova, A. P. Mathur, and K. S. Trivedi. Comparison of Architecture-Based Software Reliability Models. In *Proc. of the* $12^{th} ISSRE$. Elsevier Science, 2001.

[GPT01] K. Goševa-Popstojanova and K. S. Trivedi. Architecture-Based Approach to Reliability Assessment of Software Systems. In *Proc. of the* $12^{th} ISSRE$, pages 22–31. IEEE Computer Society, 2001.

[Gro03] Object Management Group. Uml 2.0 ocl specification. Technical report, October 2003.

[HM01] G. Huszerl and I. Majzik. Modeling and analysis of redundancy management in distributed object–oriented systems by using UML statecharts. In *Proc. of the* 27^{th} *EuroMicro Conference, Workshop on Software Process and Product Improvement, Poland*, pages 200–207, 2001.

[IEE90] IEEE Institute of Electrical and Electronics Engineers. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries, New York.* IEEE, 1990.

[ITU96] ITU. ITU-T Recommendation Z.120 Message Sequence Charts (MSC'99). Technical report, ITU Telecommunication Standardization Sector, Geneva, 1996.

[JK99] J.Magee and J. Kramer. *Concurrency: State Models and Java Programs.* John Wiley, New York, 1999.

[Kel76] R. Keller. Formal Verification of Parallel Programs. In *Communications of the ACM, 19, 7*, 1976.

[KM97] S. Krishnamurthy and A. P. Mathur. On the estimation of reliability of a software system using reliabilities of its components. In *Proceedings of the* 8^{th} *International Symposium on Software Reliability Engineering (ISSRE '97), Washington, DC, USA*, page 146. IEEE Computer Society, 1997.

[KMY91] J. P.J. Kelly, T. I. McVittie, and W. I. Yamamoto. Implementing design diversity to achieve fault tolerance. *IEEE Software*, 8(4):61–71, 1991.

[KNP04] M. Kwiatkowska, G. Norman, and D. Parker. Prism 2.0: A tool for probabilistic model checking. In *QEST '04: Proceedings of the The Quantitative Evaluation of Systems, First International Conference on (QEST'04)*, pages 322–323, Washington, DC, USA, 2004. IEEE Computer Society.

[KPP95] B. Kitchenham, L. Pickard, and S. L. Pfleeger. Case studies for method and tool evaluation. *IEEE Software*, 12(4):52–62, 1995.

[Kub89] P. Kubat. Assessing reliability of modular software. *Operations Research Letters*, 8(1):35–41, 1989.

[KWB03] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained.* Addison–Wesley Series Editors, 2003.

[LB04] Sacha Labourey and Bill Burke. Jboss clustering. Technical report, The JBoss Group, May 2004.

[LHC+05] J-H. Lo, C-Y. Huang, I-Y. Chen, S-Y. Kuo, and M. R. Lyu. Reliability assessment and sensitivity analysis of software reliability growth modeling based on software module structure. *Journal of Systems Software*, 76(1):3–13, 2005.

[Loe04] S. Loecher. Model-based transaction service configuration for component-based development. In 7^{th} *Symposium on Component-Based Software Engineering(CBSE7). Edinburgh, Scottland,*, volume 3054. LNCS, Springer, March 2004.

[LS92] K. G. Larsen and A. Skou. Compositional verification of probabilistic processes. In *CONCUR '92: Proceedings of the Third International Conference on Concurrency Theory*, pages 456–471, London, UK, 1992. Springer-Verlag.

[Lyu95] M. R. Lyu. *Software Fault Tolerance.* John Wiley & Sons, Inc., New York, NY, USA, 1995.

[Lyu96] M. R. Lyu. *Handbook of Software Reliability Engineering.* IEEE Computer Society Press and McGraw-Hill, 1996.

[Lyu07] M. R. Lyu. Software reliability engineering: A roadmap. In *FOSE '07: 2007 Future of Software Engineering*, pages 153–170, Washington, DC, USA, 2007. IEEE Computer Society.

[MA01] D. A. Menascé and V. A. F. Almeida. *Capacity Planning for the Web Services.* Prentice Hall PTR, 2001.

[MH02] I. Majzik and G. Huszerl. Towards dependability modeling of FT-CORBA architectures. In *Proc. 4^{th} EDCC, Toulouse)*, pages 121–139. Springer–Verlag, 2002.

[MIO87] J. D. Musa, A. Iannino, and K. Okumoto. *Software reliability: measurement, prediction, application.* McGraw-Hill, Inc., 1987.

[MKG97] J. Magee, J. Kramer, and D. Giannakopoulou. Analysing the behaviour of distributed software architectures: a case study. *Future Trends of Distributed Computing Systems (FTDCS '97)*, 00:240, 1997.

[MKG98] J. Magee, J. Kramer, and D. Giannakopoulou. Software architecture directed behaviour analysis. In *Proceedings of the 9^{th} International Workshop on Software Specification and Design*, page 144, Washington, DC, USA, 1998. IEEE Computer Society.

[MO83] J. Musa and K. Okumoto. Software reliability models: Concepts, classification, comparisons, and practice. *Electronic Systems Effectiveness and Life Cycle Costing*, NATO ASI Series(F3):395–424, 1983.

[Moh97] M. Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311, 1997.

[MPB03] I. Majzik, A. Pataricza, and A. Bondavalli. Stochastic Dependability Analysis of System Architecture Based on UML Models. In Rogerio de Lemos, Cristina Gacek, and Alexander Romanovsky, editors, *Architecting Dependable Systems, LNCS–2667*, pages 219–244. Springer Verlag, 2003.

[Mus79] J. D. Musa. Validity of execution time theory of software reliability. In *IEEE Transactions on Software Reliability*, volume 28(3), pages 181–191, 1979.

[Mus93] J. D. Musa. Operational profiles in software-reliability engineering. *IEEE Software*, 10(2):14–32, 1993.

[OMG00] Object Management Group. Fault Tolerant CORBA Specification. Technical report, http://www.omg.org/docs/ptc/00-04-04.pdf, April 2000.

[OMG01] Object Management Group. Model Driven Architecture. Technical report, http://cgi.omg.org/docs/ormsc/01-07-01.pdf, July 2001.

[OMG02a] Object Management Group. UML Profile for CORBA Specification. Technical report, http://www.omg.org/docs/ptc/02-04-01.pdf, April 2002.

[OMG02b] OMG. *XMI Specification.* http://www.omg.org/cgi-bin/doc?formal/2002-01-01, Jan 2002.

[OMG03a] Object Management Group. Unified Modeling Language (UML), version 1.5. Technical report, http://www.omg.org/cgi-bin/doc?formal/01-09-67.pdf, March 2003.

[OMG03b] OMG. *MOF 2.0 Specification.* http://www.omg.org/cgi-bin/doc?ptc/2003-10-04, Oct 2003.

[OMG04a] Object Management Group. *UML 2.0 Superstructure.* http://www.omg.org/cgi-bin/doc?ptc/2004-10-02, 2004.

[OMG04b] OMG. *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms.* http://www.omg.org/docs/ptc/04-09-01.pdf, Sep 2004.

[OMG05] OMG. *UML Profile for Schedulability, Performance and Time Specification.* http://www.omg.org/technology/documents/formal/schedulability.htm, Jan 2005.

[PMM93] J.H. Poore, H. D. Mills, and D. Mutchler. Planning and Certifying Software System for Reliability. In *IEEE Software*, volume 10(1), pages 88–99. IEEE, January 1993.

[PPV00] D. E. Perry, A. A. Porter, and L.G. Votta. Empirical Studies of Software Engineering: A Roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*, pages 345–355. ACM Press, April 2000.

[Ran75] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM.

[RRE04] G. Rodrigues, G. Roberts, and W. Emmerich. Reliability Support for the Model Driven Architecture. In Rogerio de Lemos, Cristina Gacek, and Alexander Romanovsky, editors, *In Architecting Dependable Systems II –LNCS 3069.* Springer Verlag, 2004.

[RRU05a] G. Rodrigues, D. Rosenblum, and S. Uchitel. Reliability prediction in model driven development. In *In Proc. of ACM/IEEE 8th International Conference on*

Model Driven Engineering Languages and Systems, LNCS 3713, pages 339 – 354. Springer, October 2005.

[RRU05b] G. Rodrigues, D. Rosenblum, and S. Uchitel. Sensitivity Analysis for a Scenario-Based Reliability Prediction Model. In *In Proc. of ICSE/WADS, St. Louis – Missouri, USA*. ACM, 2005.

[RRU05c] G. Rodrigues, D. Rosenblum, and S. Uchitel. Using Scenarios to Predict the Reliability of Concurrent Component-Based Software Systems. In *Proc. ETAPS 2005 Conference on Formal Approaches to Software Engineering*, pages 111–126. Springer, LNCS 3442, 2005.

[RRW07] G. N. Rodrigues, D. Rosenblum, and J. Wolf. Reliability analysis of concurrent systems using ltsa. In *ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 63–64, Washington, DC, USA, 2007. IEEE Computer Society.

[RSB05] E. Roman, R. P. Sriganesh, and G. Brose. *Mastering Enterprise Java Beans, Third Edition*. John Wiley & Sons, Inc, 2005.

[RSP03] R. H. Reussner, H. W. Schimdt, and I. H. Poernomo. Reliability prediction for component-based software architectures. In *The Journal of Systems and Software*, volume 66, pages 241–252, 2003.

[SCC+01] H. Singh, V. Cortellessa, B. Cukic, E. Gunel, and V. Bharadwaj. A bayesian approach to reliability prediction and assessment of component based systems. In *Proc. of the* 12^{th} *IEEE ISSRE*, pages 12–21. IEEE, 2001.

[SE01] J. Skene and W. Emmerich. A Model Driven Architecture Approach to Analysis of Non-Functional Properties of Software Architecture. In *Proc. of the* 18^{th} *ASE. Toronto, CA*. IEEE Computer Society, October 2001.

[Sho76] M. Shooman. Structural Models for Software Reliability Prediction. In *Proc. of the* 2^{nd} *ICSE*, pages 268–280, 1976.

[Som01] I. Sommerville. *Software Engineering*. Addison-Wesley, sixth edition, 2001.

[SV04] A. Sokolova and E.P. de Vink. Probabilistic automata: system types, parallel composition and comparison. In C. Baier, B.R. Haverkort, H. Hermanns, J.-P. Katoen,

and M. Siegle, editors, *Validation of Stochastic Systems: A Guide to Current Research*, pages 1–43. LNCS 2925, 2004.

[Szy02] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[Tra95] C. Trammell. Quantifying the reliability of software: statistical testing based on a usage model. In *ISESS '95: Proceedings of the 2nd IEEE Software Engineering Standards Symposium*, page 208, Washington, DC, USA, 1995. IEEE Computer Society.

[TU06] C-C. Tan and S. Uchitel. Determinisation of Generative Probabilistic Labelled Transition Systems. Private communication, 2006.

[UCKM03] S. Uchitel, R. Chatley, J. Kramer, and J. Magee. LTSA-MSC: Tool Support for Behaviour Model Elaboration Using Implied Scenarios. In *Proc. of* 9^{th} *TACAS, Warsaw*, April 2003.

[UCL07] UCL Research Computing. The UCL Condor Pool. Technical report, http://www.ucl.ac.uk/research-computing/services/condor/index.html, 2007.

[UKM03] S. Uchitel, J. Kramer, and J. Magee. Synthesis on Behavioral Models from Scenarios. In *IEEE Transactions on Software Engineering*, volume 29(2), pages 99–115. IEEE, February 2003.

[UKM04] S. Uchitel, J. Kramer, and J. Magee. Incremental Elaboration of Scenarios-Based Specifications and Behavior Models Using Implied Scenarios. In *ACM Transactions on Software Engineering and Methodologies*, volume 13(1), pages 37–85. ACM Press, January 2004.

[Uni06] University of Wisconsin-Madison. Condor High Throughput Computing Manual Pages, version 6.7.20. Technical report, http://www.cs.wisc.edu/condor/manual/v6.7/, June 2006.

[W3C99] W3C. *XSL Transformations (XSLT)*. http://www.w3.org/TR/xslt, November 1999.

[WK03] J. Warmer and A. Kleppe. *The Object Constraint Language*. Pearson Education, second edition, 2003.

[WWC99] W. L. Wang, Y. Wu, and M. H. Chen. An Architecture-Based Software Reliability Model. In *Proc. Pacific Rim International Symposium on Dependable Computing. Washington, DC , USA*, pages 143–150. IEEE Computer Society, 1999.

[XW95] M. Xie and C. Wohlin. An additive reliability model for the analysis of modular failure data. In *Proceedings of the* 6^{th} *International Symposium on Software Reliability Engineering (ISSRE '95)*, pages 188–194. IEEE Computer Society, 1995.

[YCA99] S. M. Yacoub, B. Cukic, and H. H. Ammar. Scenario-Based Reliability Analysis of Component-Based Software. In *Proc. of the* 10^{th} *ISSRE, Boca Raton, FL, USA*. IEEE, November 1999.