NOTE TO USERS

This reproduction is the best copy available.

UMI[®]

Dependability Modelling of Layered Systems

by

Olivia Das, M. Sc.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Ottawa-Carleton Institute for Electrical and Computer Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada K1S 5B6
February 2004

Copyright © 2004, Olivia Das

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisitons et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-94192-2

Our file Notre référence

ISBN: 0-612-94192-2

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this dissertation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de ce manuscrit.

While these forms may be included in the document page count, their removal does not represent any loss of content from the dissertation.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

Most distributed systems with clients and servers are constructed with a layered software architecture. In such layered systems, failure of a processor or a failure of a process in one layer can cause many other processes that depend on its services to fail, unless they can detect the failure and reconfigure to use a redundant process. The complexity of such systems has stimulated the use of a separate fault management architecture for automatic detection of software and hardware failures and reconfiguration. Since failures of such systems may incur high cost or may result in loss of life, analyzing the dependability of such systems is of major importance.

This thesis has introduced a new model, called the Dependable-LQN model, for evaluating the dependability of such layered systems. The model includes the management components and their failures and interactions in the analysis. An efficient model solution strategy is described, to compute the steady-state and the instantaneous measures. The strategy exploits the idea of aggregating the states of a system into multiple classes so that all the states in a class have the same set of effective components (i.e. the components which are working and are in use). This strategy avoids the generation and solution of large Markovian models that is commonly used for dependability evaluation. The scalability of the model has been demonstrated by undertaking a case study of a large-scale Air Traffic Control system.

Acknowledgments

My sincere appreciation to all those people who gave me the spirit to begin, conduct and finish this thesis.

First, I would like to express my heartfelt thanks to my supervisor, Professor C. Murray Woodside, for giving me the opportunity to do this research and for providing me with so much of inspiration. I am grateful to him for supporting me at every stage of my research, and for giving me the exposures that have shaped my career. His valuable guidance has been extremely helpful to bring success to this work. I am privileged to have this rare opportunity to learn from one of the best, kindest and the most knowledgeable Professor.

I would like to thank my external examiner, Professor Boudewijn R. Haverkort, and the examining board members for carefully reading my thesis and providing me with valuable suggestions.

Many thanks to the members of our research group for valuable discussions.

My special thanks to Daren Russ and Dave Sword for helping me retrieve my thesis data when my computer crashed.

I would also like to thank Nortel Networks, Natural Sciences and Engineering Research Council of Canada and Carleton University for their financial support.

My sincere thanks to Arindam, without whose endless encouragement and love this journey would never have started nor finished. Whenever the going got tough, he was there ready with his humor and liveliness to cheer me up. He has supported me in so many ways during this journey. He has been eagerly waiting for this work for a long time.

I am grateful to my parents, Mahamaya Pal (Mummum) and Prabir Kumar Pal (Bapi), who raised me with a lot of love and care. My wholehearted thanks to Mummum without whose patience and support I would not have reached this point in my life. My father has not been given chance to witness this day. Had he been here to see this successful endeavor, he would have been so happy.

I am thankful to my Dada and Didi for their well wishes.

I dedicate this thesis to the memory of my Bapi.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Thesis argument.	4
1.3	Contributions	6
1.4	Organization of the thesis	7
2	Background	8
2.1	Basic Concepts of Fault Tolerance	8
2.2	Software Fault Tolerance.	12
2.2.1	Organization of Fault Management	14
2.3	Architectures for Fault Management.	16
2.3.1	Manager-Agent Model	17
2.4	Dependability Modelling Techniques	19
2.4.1	Reliability and Availability Modelling Techniques	20
2.4.2	Use of Architecture in Models	24
2.5	Performability Modelling Techniques.	26
2.5.1	Monolithic Models	26
2.5.2	Hierarchical Models	28
3	The Dependable-LQN Model	35
3.1	The Dependable-LQN Model Definition.	35
3.1.1	Failure behavior of a single independent component.	35
3.1.2	First part of Dependable-LQN model: FTLQN model.	36
3.1.3	Second part of Dependable-LQN model: MAMA model	42
3.2	Modelling Knowledge Propagation in MAMA.	45
3.3	Modelling Fault Propagation	49
3.3.1	Fault Propagation Graph Definition	49
3.3.2	Notation for Operational Configurations	56

3.4	Measures of Interest.....	57
3.5	Summary	58
4	The Dependable-LQN Model Solution	59
4.1	The Dependable-LQN Performability Algorithm	59
4.1.1	Example	64
4.2	Scalability Analysis (Comparison of Two Approaches).....	67
4.3	A Comparison of Four Fault Management Architectures	70
4.3.1	Model Parameters.....	71
4.3.2	Four sample fault management architectures.....	72
4.3.3	Results	77
4.4	Best Placement of Managers	80
4.5	Summary	87
5	The Dependable-LQN Model Solver	89
5.1	Description of the DependableLQNS tool.....	89
5.2	Running the tool from the command-line	90
5.3	Running the tool using the Graphical User Interface	92
5.4	Summary	99
6	Case Study: An Air Traffic Control System	100
6.1	Dependable-LQN Model of an ATC En Route System	100
6.2	Numerical Results	105
6.2.1	Parameterizing the model.....	106
6.2.2	Results on Computational Effort	106
6.2.3	Effect of management component failures on System Availability...	107
6.2.4	Sensitivity of System Availability to all components	108
6.2.5	Optimal Replication Number for Name Servers	110
6.2.6	Effect of Replication of Unreliable Agents (i.e. the gSAM servers) on System Availability.....	111
6.3	Summary	111

7	Model Extensions	114
7.1	Extension 1: Effect of Aging and Software Rejuvenation.	114
7.1.1	Example	117
7.1.2	Explosion of Operational Configurations.	119
7.2	Extension 2: Effect of Delays in Detection and Reconfiguration	120
7.2.1	Expressing Detection and Reconfiguration	120
7.3	Summary	126
8	Conclusions	127
8.1	Research Summary	127
8.2	Limitations of this thesis	129
8.3	Overall Conclusions.	130
8.4	Future Research	130
	Bibliography	131
A	BNF Description of MAMA-dl language	145

List of Figures

1.1	The class of systems considered in this thesis	3
1.2	The idea of aggregation utilized in this thesis	4
2.1	The failure-repair cycle [GR93].	9
2.2	Software fault classification and techniques for fault tolerance	12
2.3	Management components and relationships. Client, Server1 and Server2 are application processes	18
2.4	Reliability/Availability Models	20
2.5	An example of a non-coherent fault tree with basic events P1-P4	22
2.6	Performability Models	27
2.7	An example LQN model	33
3.1	Markov model for each component	36
3.2	FTLQN notations	37
3.3	An FTLQN model	38
3.4	Modelling dependent failures with AND-OR depends relationships	40
3.5	A shared software factor introduced as a pseudocomponent.	41
3.6	Separating the failure influences of the operating system and the hardware for a processor. The two factors (shown as pseudocomponents) have different failure and repair rates	42
3.7	The manager-agent model	43
3.8	MAMA notations	45
3.9	MAMA model for a centralized management architecture for layered application of Figure 3.3	46
3.10	The Knowledge Propagation graph corresponding to the MAMA model shown in Figure 3.9.	48
3.11	The Non-Coherent Fault-propagation graph corresponding to the FTLQN model of Figure 3.3 and Knowledge Propagation Graph of Figure 3.10. $know_{c,t}$ functions	

are not expanded here for clarity. Each $know_{c,t}$ would be an OR node with AND children (one child for each augmented minpath P_q^+)).	55
4.1 Non-coherent fault-tree corresponding to the operational configuration that implies “ <i>dbService is using the Server Database-B</i> ” obtained from the fault propagation graph in Figure 3.11 (Chapter 3). Each dotted enclosure realizes a <i>know</i> function. The repeated basic events are shown in italics	63
4.2 A three-tiered client-server system and its associated central management system (with agents in (b) and without agents in (c))	68
4.3 One stand-by server throughout.	69
4.4 Two stand-by servers throughout.	69
4.5 A layered model of a client-server system with two groups of users. Server2 is the backup of Server1	71
4.6 MAMA Model of a centralized management architecture for the layered application in Figure 4.5.	73
4.7 MAMA Model of a distributed management architecture for the application in Figure 4.5. dm1 and dm2 are two peer domain managers	76
4.8 MAMA Model of a hierarchical management architecture for the application in Figure 4.5. mom1 is manager of managers handling both dm1 and dm2.	77
4.9 MAMA Model of a network management architecture for the application in Figure 4.5. im1, im2 are integrated managers.	78
4.10 Comparison of expected steady state reward rate of the system in Figure 4.5 for the four management architectures	79
4.11 Shorthand notations for the deployment of agents, and for the connectors, as used in Figures 4.12-4.19.	82
4.12 Case-1: Placement of the managers m1 and m2 in callers’ processors proc1 and proc2. m1 manages Domain-1 and m2 manages Domain-2	83
4.13 Case-2: Placement of the managers m1 in proc1 and m2 in proc3 (i.e. the primary server’s processor).	83
4.14 Case-3: Placement of the managers m1 in proc1 and m2 in proc4 (i.e. the backup	

server's processor)	83
4.15 Case-4: Placement of the managers m1 in proc2 and m2 in proc3 (variation of Figure 4.13).	83
4.16 Case-5: Placement of the managers m1 in proc2 and m2 in proc4 (variation of Figure 4.14).	84
4.17 Case-6: Placement of the managers m1 and m2 on their dedicated processors. Domains are setup horizontally	84
4.18 Case-7: Placement of the managers m1 and m2 on their dedicated processors. Domains are setup vertically	84
4.19 Case 8: Placement of the managers m1 and m2 on their dedicated processors. m1 is managing the same set of components as being managed by m2	85
5.1 Flow of processing of the DependableLQNS tool.	91
5.2 Dependable-LQN model of a simple example.	93
5.3 Input file <i>simple-example.dlqn</i> corresponding to the model in Figure 5.2	94
5.4 Output generated by <i>dlqns</i> for the input in Figure 5.3	95
5.5 The GUI window	95
5.6 Opening an existing file	95
5.7 The file <i>simple-example.dlqn</i> is opened.	96
5.8 Editing "Processor2"	97
5.9 Editing connector "c1"	97
5.10 Editing "Task1"	97
5.11 Editing "Entry1"	97
5.12 Editing "service2"	98
5.13 Editing the management task "Agent1"	98
5.14 Adding a processor	98
5.15 The Results window	99
6.1 A Dependable-LQN model for an ATC en route system. The redundant server groups are not shown here. The dotted boundary indicates one replica of a replicated subsystem	102

6.2	Portion of MAMA model for Figure 6.1. Communication between the tasks DisplayManagement and the replicas of FlightPlanManagement is made transparent by the P2PSM task residing in the consoleProc	103
6.3	Portion of MAMA model for Figure 6.1. Redundant server groups and interactions among the gSAM servers in a group are not shown here	104
6.4	Portion of MAMA model for Figure 6.1 showing interactions among the gSAM servers in a group of two replicas of the central subsystem	105
6.5	The predicted System Availability as a function of the availabilities of the management components.	108
6.6	Part of the MAMA model showing the interactions in the Central subsystem group. There are two replicas of gSAM servers in each processor.	112
7.1	CTMC model for each processor.	116
7.2	CTMC model for each software task.	116
7.3	A simple client-server system	118
7.4	An example of a Dependable-LQN model	122
7.5	Detection-reconfiguration graph for the system in Figure 7.4 in response to the failure of Task 3, where all components were operational at the time of failure .	123
7.6	Detection-reconfiguration graph for the system in Figure 7.4 in response to the failure of Task 6 where all components were operational at the time of failure. .	124
7.7	CTMC model for each component c	125

List of Tables

4.1 Standby-Redundancy policy: Distinct Operational Configurations and their probabilities obtained using State Aggregation Approach	65
4.2 Load-balanced policy: Distinct Operational Configurations and their probabilities obtained using State Aggregation Approach	66
4.3 Computation time for determining the distinct operational configurations by the Enumeration Approach and the State Aggregation Approach for the models in Figures 4.2 to 4.4	70
4.4 Configuration Probabilities (for Centralized Management) and Rewards	74
4.5 Distinct Operational Configurations of the system in Figure 4.5, their probabilities for four fault management architectures and the associated throughput of two groups of users	79
4.6 Probabilities of Operational Configurations of the system in Figure 4.5 for different manager placements and the associated throughput of two groups of users (UserA and UserB)	86
6.1 Sensitivity of System Availability to availabilities of components	109
6.2 System Availability for different number of NameServer replicas	110
6.3 System availability with or without replicated gSAM servers	111
7.1 Probabilities and Rewards for Operational Configurations	119

Chapter 1

Introduction

1.1 Motivation

In recent years, there has been a widespread usage of large and complex fault-tolerant distributed systems which can continue providing service after the occurrence of failures. The fault-tolerance in these systems may be achieved by replicating the system components and by employing mechanisms for failure detection and reconfiguration. Examples of such systems include aircraft control systems, telecommunication systems, banking systems and airline reservation systems.

In order to ensure correct and timely operation of the type of systems mentioned above, it is useful to model them to evaluate dependability measures, and compare them to requirements. *Dependability* [Lapr85] is the trustworthiness of a system such that reliance can justifiably be placed on the service it delivers. The three most important attributes of dependability are reliability, availability and performability. Reliability and availability are based on the failure and repair behavior of the components and the structure of the system. On the other hand, *performability* [Meye80] combines the failure-repair aspects with the performance behavior of the system which is concerned with the contention for system resources. The concept of performability applies to systems that might be restructured in response to faults and may continue to perform some useful work with a degraded performance. This thesis uses the measures of performability.

The common approach to performability modelling has been the use of Markov Reward Models in which the failure-repair aspects of the system are modelled by a Continuous-time Markov chain and a reward rate (measure of performance) is associated with each of the states of the Markov chain. The performability measure is then evaluated by combining the reward rates associated with the states of the Markov model with the state probabilities. For large realistic systems, the state space in a Markov model becomes very

large so that the generation, storage and solution of such models becomes difficult.

Since a reward rate is attached to the various states of the Markov model, it is possible to partition the state space into equivalence classes, each class being represented by an aggregated state which is defined as the set of states that have the same performance model and thus have the same value of the reward rate. The efficiency in the performability calculation therefore lies in devising a method that would determine these aggregated states, thereby minimizing the number of performance solutions required and compute the probabilities of these states without solving the large Markov model and without generating/enumerating all the states of the system. This thesis uses aggregations based on operational configurations of a layered performance model.

Nowadays, most of the distributed systems with clients and servers are typically constructed with a layered software architecture (which in this work simply means subsystems in service layers, that offer services to other subsystems in higher layers). Layered architectures are popular for distributed systems such as information processing systems, web applications and database systems. Compared to centralized computing they enhance the system usability, scalability and availability. In these layered applications, failure of a processor or a failure of a process in one layer can cause many other processes that depend on its services to fail, unless they can detect the failure and reconfigure to use a redundant process. More often, these systems use a separate fault management architecture for monitoring and controlling of the status of the application software and hardware (instead of handling the faults within the application itself). Such usage promotes software reuse and also eases the development of the software application.

Consideration of a separate management architecture influences the fault coverage of the layered system where *fault coverage* measures the ability of the system to recover. This is because the system's knowledge about failures in order to recover from them now depends on the status of management components and also on the management architecture. Success at a reconfiguration point now depends on the system structure and the connectivity between the source of failure and the point of reconfiguration in the fault

management architecture. In fact, a failure may be fully, partially or not covered at all by the system, depending on how many of the necessary reconfigurations are successful. When a failure is partially covered, the system may end up failed or may operate with degraded performance, compared to the fully covered case. Thus, it is necessary to consider the effect of partial coverage of a failure for full performability analysis of layered systems. Some recent work [GWTH98], [GT01] has considered the effects of software architecture in reliability modelling, however it did not consider a management architecture, or its related failures. Analysis of these layered systems that considers their fault management architecture into account would be of great importance.

Figure 1.1 The class of systems considered in this thesis

This thesis aims in developing a performability model for the class of systems mentioned above and is shown in Figure 1.1. It provides an efficient model analysis technique that determines the various operational configurations of the system (where each operational configuration corresponds to a distinct performance model) and compute their probabilities without solving very large Markovian models. The idea of aggregation is presented in Figure 1.2. In this figure, an *effective* component means a component which is working and is in use by the system.

Figure 1.2 The idea of aggregation utilized in this thesis

1.2 Thesis argument

A model for evaluating performability for layered systems was first outlined by Dr. C. M. Woodside in [Wood96]. It was called Fault-Tolerant Layered Queueing Network (FTLQN) model in [DW01a] that provided an efficient algorithm for computing steady-state measures from this model. However, these studies assume instantaneous perfect detection and reconfiguration, independent failures and repairs and only primary-standby replication strategy to obtain fault-tolerance.

This thesis has developed a model, termed as *Dependable-LQN* model, that

- extends the FTLQN model, by incorporating:
 - pseudo-components to model some kinds of failure dependency between components.
 - other replication mechanisms such as load-balancing replication, active replication and primary-standby-active replication.
- captures a management architecture and its subsystem failures. The architecture model developed for describing various fault management architectures is termed as MAMA model (*Model for Availability Management Architectures*).

This thesis has described an efficient algorithm that determines the distinct classes of system states that have the same performance model, termed as *operational configurations*, and computes their probabilities without solving large Markov models. It first determines the connectivity between a management element that detects a failure and a point of reconfiguration, by applying minpath algorithms to a *Knowledge Propagation Graph* (a directed graph, obtained from the MAMA model, that captures the connectivity information of the management architecture). This is then used in a *Fault Propagation Graph*, which is an AND-OR-NOT graph that captures the service dependencies of the FTLQN model, to give operational configurations and probabilities. A kind of non-coherence is introduced into the Fault Propagation Graph because the occurrence of a failure does not by itself trigger a reconfiguration; the knowledge of the failure must reach the decision-making elements. This makes it an AND-OR-NOT graph, instead of an AND-OR graph as used in [DW01a] (which assumed perfect knowledge). To compute the probabilities of the operational configurations, this work uses non-coherent fault trees [DR97]. The fault propagation graph can be constructed so as to model the primary-standby redundancy (redirect requests to a standby server), load-balancing redundancy (redistribute the load among the remaining operational servers), active redundancy (workload sent to all the operational servers) or primary-standby-active redundancy (load sent to all operational servers but only the primary would send further requests to lower layers of servers).

To give a performability measure, the following steps are used:

- generate a Knowledge Propagation graph (from the MAMA model) and apply min-path algorithms to the graph to obtain connectivity information.
- generate an AND-OR-NOT graph, called the Fault-Propagation graph, from the FTLQN model and the connectivity information of the Knowledge Propagation graph
- find the different operational configurations of the system from the fault propagation graph
- compute the probability of each operational configuration (using a non-coherent fault tree)

- determine the reward rate for each operational configuration (using a Layered Queueing model for each one)
- combine the probabilities and the rewards to determine the steady-state performability or point-performability.

The Dependable-LQN model can be seen as an extension of the Dynamic Queueing Analysis approach of Haverkort et.al. [HNZ92], [Have95]. The extensions include layered queueing analysis in place of standard flat queueing models, and the role of layered service dependencies in propagating failures. The calculations are highly scalable, essentially because a large number of possible failure states often lead to just a few distinct performance models to be solved due to layered dependencies.

As part of the thesis, a computational tool has been developed to automate the Dependable-LQN model solution that uses the Aralia software for non-coherent fault trees [DR97] and the Layered Queueing Network Solver (LQNS) [FMNPRW96] for the performance calculations for each operational configuration.

This thesis also presents two further extensions to the Dependable-LQN model and outline solutions for the extensions. The extensions are consideration of aging for components with the effect of software rejuvenation, and delays due to detection and reconfiguration.

1.3 Contributions

The major contributions of this thesis are as follows:

- *Development of a performability model:* A model, called *Dependable-LQN*, has been developed (Chapter 3) for layered systems that employs a separate management architecture for failure detection and reconfiguration. This includes a management architecture model MAMA and an architecture language MAMA-*dl*.
- *Development of an efficient solution algorithm for the Dependable-LQN model:* A new solution algorithm has been developed (Chapter 4) for the Dependable-LQN model, as outlined in the previous section.

- *Development of a tool to support the modelling technique:* A software tool termed as DependableLQNS, the *Dependable-LQN Model Solver*, has been developed (Chapter 5) to automate the model analysis.
- *Demonstration of the model on a large system:* A model for a large-scale Air Traffic Control system is constructed and analyzed (Chapter 6) in order to demonstrate the model's capability in evaluating large systems.
- *Extensions of the model:* The model has been extended in two directions (Chapter 7):
 - the effect of aging of components and of software rejuvenation
 - the effect of delays in detection and reconfiguration

The model solution has been outlined for both the cases.

1.4 Organization of the thesis

Chapter 2 provides the background material for the topics which are utilized in the development of this thesis. Chapter 3 describes the Dependable-LQN model concept. Chapter 4 explains the model solution algorithm. Chapter 5 presents a high-level description of the solver. Chapter 6 presents the results obtained by applying the model to a large Air Traffic Control system in order to demonstrate the scalability of the model. Chapter 7 describes the two extensions for the model. Chapter 8 summarizes this research work and discusses future research that could be built upon this work.

Chapter 2

Background

The area of fault tolerance and dependability modelling is extremely wide and this chapter gives only an overview of the parts that are most essential for background for the thesis. It discusses software fault tolerance, architectural approaches to management of failures and existing analytical dependability modelling techniques.

2.1 Basic Concepts of Fault Tolerance

Fault, Error and Failure

A system is regarded as being composed of a number of components or subsystems, with interactions among them. Each of the subsystems may have an internal structure consisting of smaller subsystems. This hierarchy may continue until it is not necessary to further specify the details of the system. Subsystems at this last level are called “atomic components”. If we are only interested in major components like processors, disks and software, then our hierarchy need not look at the detailed structure of gates and chips. The following definitions discuss the behavior of a single component, however the terminology applies recursively to subsystems and systems.

Each component has a *specified behavior* and an *observed behavior* [GR93]. A *failure* occurs when the observed behavior of a component deviates from its specified behavior, i.e. a component fails when it cannot provide the desired service. A failure occurs because of an *error*, or a defect in the component. The cause of the error is a *fault*. When an error causes a failure, it becomes *effective*, before that it was *latent*.

An example of the above definitions is: a programmer’s mistake is a fault. It induces a latent error in the software. When the erroneous instructions of the software execute, they produce incorrect results and may cause a failure.

The observed behavior of a component alternates between the states, “Service Accom-

plishment” and “Service Interruption”. These states are illustrated in Figure 2.1 [GR93]. A component’s observed behavior matches its specified behavior in *Service Accomplishment* state. Sometimes, a fault causes an error that eventually becomes effective resulting in the component’s failure. Then, the component enters the *Service Interruption* state. The failure is detected, reported, corrected or repaired and then the component returns to Service Accomplishment state.

Figure 2.1 The failure-repair cycle [GR93]

Fault Classification

Faults can be classified according to their duration as transient or permanent. The duration specifies the length of time the fault is active. *Permanent* faults are those in which once the component fails, it remains in that state indefinitely until some corrective actions are taken. *Transient* faults are of short durations, caused by temporary malfunction of the system or due to some external influence. Transient faults can cause a failure only in the duration for which they exist.

Faults can also be classified based on how a component behaves when it fails. [CAS86] classifies faults in four categories: crash, omission, timing and byzantine.

- A *crash* fault is a fault that causes the component to halt or to lose its internal state. A

special case is a *fail-stop* fault [Schn93] (considered in this work) where the failure is detectable by other components.

- An *omission* fault is a fault that causes a component to not respond to some inputs.
- A *timing* fault is a fault that causes a component to respond too early or too late.
- A *byzantine* fault is a fault that causes the component to behave in an arbitrary manner.

Techniques to deal with faults

There are mainly three techniques: fault avoidance, fault removal and fault tolerance. *Fault avoidance* attempts to prevent the introduction of faults. It can include things such as design reviews, code inspection, structured programming, software reuse and formal methods. *Fault removal* is a technique that finds and removes faults already introduced. It can include things such as testing, verification and validation. *Fault tolerance* is an execution-time technique that cope with the faults, and is the technique considered here.

Fault Tolerance

The goal of fault tolerance is to continue providing service after the occurrence of failures. There are two approaches to provide it: fault masking and reconfiguration. *Fault masking* completely masks or hides the failures by using the redundancy present in the system. In this case, detection of failures and recovery activities are not needed. *Reconfiguration*, on the other hand, is the process of eliminating a faulty component from a system and restoring the system to some operational state. In this case, four activities are carried out: failure detection, failure location, failure containment and failure recovery. *Failure detection* is the phase of identifying that a failure has occurred. *Failure location* is the process of identifying where a failure has occurred. *Failure containment* is the process of isolating the failure so its effects cannot propagate. *Failure recovery* is the process of regaining operation via reconfiguration.

Fault tolerance can be applied at various levels in a computer system. Hardware fault

tolerance is provided using Triple-Modular Redundancy or other techniques [Prad86], [Jalo94]. In the operating system and database layers, fault tolerance is often provided using replicated file systems [SKKOSS90], disk shadowing [BG88], transaction-based checkpointing and recovery [NF92] and other system routines. These methods handle failures occurring in the underlying hardware or the operating and database system layers only. Many studies have recognized that computer system outages are more due to application software failures than to hardware failures [GS91], [SC91]. Software fault tolerance techniques are described in Section 2.2.

Defining the Objectives of Fault-Tolerance

The main objective is to meet the dependability requirements of a system. Dependability is defined as the trustworthiness of a computer system such that reliance can justifiably be placed on the service it delivers [Lapr85], [Lapr92]. The three most significant dependability attributes are: reliability, availability and performability.

Reliability deals with continuity of service and it is a function of time, $R(t)$, defined as the conditional probability that the system performs correctly throughout the interval of time $[0, t]$, given that the system was working correctly at time $t=0$. Examples of highly reliable applications are aircraft flight control and space applications. *Availability* deals with readiness of usage and it is a function of time, $A(t)$, defined as the probability that a system is operating correctly and is available to perform its functions at instant of time, t . Examples of highly available applications are banking systems and airline reservation systems. If a system is not repairable, the definition of $A(t)$ is equivalent to that of $R(t)$. They differ in that reliability involves an interval of time whereas availability is taken at instant of time. *Performability* [Meye80] combines the fault-tolerance, reconfiguration and failure-repair aspects of system behavior with its performance behavior (which is concerned with contention for system resources). The goal of performability is to capture the overall responsiveness of a system which is sometimes degraded by failures. It considers graceful degradation of systems where if a component fails, the system itself detects the failure and automatically reconfigures, reaching a degraded state of operation in which it continues to

provide service but at a reduced capacity [Losq77]. A degradable system can have several reduced operational states (each state providing some performance level) between fully operational and completely failed.

2.2 Software Fault Tolerance

Software is not a physical entity like hardware and so the only kind of faults it has are design and coding faults. This section describes the various types of software design faults, the fault-tolerant techniques to deal with those faults, organization of fault management and the detection and recovery mechanisms.

Software fault classification and fault-tolerance techniques

There are three kinds of software faults: Bohrbugs, Heisenbugs and Aging-related bugs. Figure 2.2 shows the fault classification and the techniques used for fault-tolerance.

Figure 2.2 Software fault classification and techniques for fault tolerance

Bohrbugs [Gray86] are the faults that are easily reproducible and cause permanent failures. They can be easily identified and removed during the testing and debugging

phase of the software life cycle. To deal with these bugs at run-time, design diversity techniques and data diversity techniques were developed. In design diversity, if a software module cannot provide service, then another module which has a different design is used to provide the desired service. The three well known techniques for design diversity are N-version programming introduced in [AC77], recovery-block approach first described in [HLMR74] and N-self checking programming [LABK90]. Data Diversity [AK87], on the other hand, uses one version of software and it relies on the observation that a software sometimes fails for certain values in the input space and this failure could be averted if there is a minor change of input data which is acceptable to the software. N-copy programming [AK87] is a well known technique based on data diversity, where different copies of the program are executed in parallel with different input data sets.

Heisenbugs [Gray86] are the faults that cause transient failures, i.e. the failure may not recur if the software is restarted. These are the bugs which are revealed only during specific interaction of events. Some studies [SC91] have reported that a large proportion of software failures are transient in nature caused by timing errors, workload and exception handling. The study of failure data from Tandem's fault tolerant computer system indicated that 70% of the failures were transient failures, caused by faults like race conditions and timing problems [LI95]. Heisenbugs are seen more often in distributed systems than in uniprocess systems. This is because the behavior of a distributed system, especially a client-server application, not only depends on the input data and message contents but also on the timing and interleaving of messages and the shared variables [Cris91]. Although design and data diversity techniques can be used to deal with Heisenbugs, however, more efficient techniques are to retry a failed operation, or restart a crashed process automatically or reboot a processor. For doing this, fault detection capabilities are required.

Aging-related bugs [HKKF95] are the faults that cause deterioration of the operating system resources, data corruption and numerical error accumulation. Examples include memory leaks, unreleased file locks, storage space fragmentation, accumulation of round-off errors etc. The fault conditions gradually accumulate over time and lead to performance degradation of the software or transient failures or both. There are two main

approaches for fault-tolerance in this case: reactive and proactive. The reactive approach include restarting the software on failure or rebooting a node. Proactive approach, also called *software rejuvenation* [HKKF95], involves occasionally halting the software, cleaning its internal state and restarting it. Some common examples of software rejuvenation are garbage collection, defragmentation and flushing kernel tables; a more drastic example is rebooting the operating system.

2.2.1 Organization of Fault Management

There are three ways to organize software failure detection and management.

First, it can be provided in the lower layers of the software, or in hardware. For example, it is possible to detect a software failure and restart the application at a checkpointed state through the operating system utilities, as in IBM's MVS [SW98]. However, [SRC84] discussed that such methods of fault management may be of little value when compared with the cost of providing them at these lower layers. They described that fault tolerance cannot be complete without the knowledge and help from the endpoints of an application, i.e. the application software has to be engaged to provide a complete end-to-end fault tolerance.

Second, it can be provided by the application layer. It has been argued by [HK95] that detection is better at the application layer than in the lower layers. For example, replicating only the critical files of the application (which are only known by the application) at the application layer is better in execution time and space than the file replication in a mirrored disk using operating system facilities. This concept of providing fault detection and recovery capabilities in the application layer can not only be used for handling application software failures but also for underlying hardware and operating system failures if they are undetected in those layers.

A third approach (considered in this work) is to use a separate software subsystem that provides the fault-tolerant functionalities. Such usage removes the intricacies of the failure detection mechanisms from the application, and avoids repeating them in every program.

Tandem's NCAPS [Lara98] provides specialized software for detection and recovery for UNIX clusters. Huang et al. [HCKLW98] describes a set of components called "NT-swift" for building fault-tolerant applications in Windows NT. [SFKLL98] describes an external failure detection service for grid applications. Various architectures are described in Section 2.3. It has also been proved [CT96] that using a separate failure detection service can also circumvent the impossibility result [FLP85] in distributed systems which says "some basic reliable distributed algorithms, such as Consensus, cannot be solved in an asynchronous distributed system subject to crash failures". Such usage of a separate management system impacts the failure coverage of the application since the failures of these management components hamper the detection capabilities and thus the application recovery. Hence it necessitates the need to consider the management architecture and its failures into the system analysis.

Mechanisms for Failure Detection

Crash failures of components can be detected by mechanisms such as heartbeats, time-outs on periodic polls, and time-outs on requests between application processes. Heartbeat messages from an application process can be generated by a special heartbeat interrupt service routine which sends a message to a local *agent*, every time an interrupt occurs, as long as the process has not crashed or hung. Heartbeat messages for an entire processor can be generated by an agent configured similarly, to show that the processor is functioning. A component that cannot initiate heartbeat messages may be able to respond to messages from an agent; we can think of these as status polls. The responses give the same information as heartbeat messages. Polls to a processor could be implemented as pings, for instance. In either case the agent listens for a message and assumes failure if the delay is too long. [SFKLL98] describes "heartbeat monitors" that are like agents and they use heartbeat mechanism to detect failures. [HCKLW98] has "watchdog daemon" processes that are like agents and they also use heartbeat or polling mechanisms to detect application failures and use pings to detect processor failures.

Mechanisms for Failure Recovery

A failure is usually recovered by use of replication mechanisms. The purpose of replication is to provide redundant copies of a process so that a process can continue to provide useful service even if some of its replica fails or the processors hosting some of the replicas fails.

There are mainly two kinds of replication techniques: passive (also called primary-backup) and active replication [Nara99]. In *passive* replication, there is a designated primary that receives the requests, performs all the operations and returns the response, and when the primary fails, one of the backups is promoted to become primary. There can be two variants (cold or warm) of this replication for stateful systems depending on the degree to which the states of the backup replicas lag behind the primary replica. For *cold passive* replication, the backup is loaded in the memory only in the event of failure of the primary replica. In this case, the recovery time is high because of the initialization and state-transfer procedures for the backup replica using checkpointing and logging mechanisms. For *warm passive* replication, all the backup replicas are created and initialized and the primary sends periodic state updates to the backups. In this case, the recovery time is lower than the cold scheme, however, the runtime overhead is high. In *active* replication, all the replicas receive the requests, perform operations and return the response. All replicas are assumed to have a deterministic behavior and require an atomic broadcast mechanism to synchronize messages. In this case, the computation continues as long as one replica is running. A variant of this replication is the *load-balancing* approach where the workload is distributed equally among the replicas.

2.3 Architectures for Fault Management

A fault management system controls and monitors the health of a distributed application. It performs activities that include failure detection, fault isolation, and automatic recovery from failures by reconfiguring to use redundant application components.

Fault management architecture represents a high-level abstraction of a fault manage-

ment system. It captures information about the management components, their roles and how those components are interconnected. There are three reasons why management architectures are important: a) it helps in mutual communication between developers, b) it captures the early design decisions that not only affect the management system's quality attributes but also influence the application's quality attributes, c) it can be applied to other management systems with similar requirements and can promote large scale reuse. This section is going to describe the basic model, called Manager-Agent model, used by most management systems.

2.3.1 Manager-Agent Model

The management components and relationships for the Manager-Agent model are indicated in Figure 2.3, following [Kreg01]. Agents mediate between application components and managers. Each application processor usually runs an agent that monitors all the application processes running on that processor and also monitors the operating system's health status. An agent is responsible for sending events to the manager, relaying data and command requests from the manager to the application processes, collecting responses and forwarding them to the requester. There are one or more managers which are responsible for collecting status information from the agents (using a management protocol such as SNMP or JMX), making decisions and issuing notifications for reconfiguration.

Managed application processes have embedded modules called "Subagents" which have a set of responsibilities that include exposing an appropriate interface to the management system, sending heartbeat messages to the local agent, or responding to requests from an agent or a manager. Reconfiguration can be handled by a subagent who might cause a process or an Object Request Broker (ORB) to retarget its service requests. It can also be handled by an agent that might restart a process, or reboot a processor altogether.

The agents and managers are described here as if they are free-standing processes, even though in practice some of these components may be combined with other components in a fault-tolerant ORB [FGS98], [MMN98], or an application management system such as Tivoli [Tivoli].

Figure 2.3 Management components and relationships. Client, Server1 and Server2 are application processes

This manager-agent model covers several other models described in the literature. For example, the Tandem’s NCAPS architecture described in [Lara98] has a “node status monitor” (NSM) on each node and they exchange heartbeats to detect node failures. There is also a “process pair manager” (PPM) on each node which manages the application processes on that node and also exchange messages with other PPMs to know about the status of the remote application processes. Each PPM registers to the NSM on its node to get notified about the remote node failures. For this model, we can think of the PPMs as managers and the NSMs as agents. In this case, managers are communicating with the agents and other managers directly.

Based on the manager-agent model, various different architectures are being used. Following is a classification of architectures given in [SRM95]:

- centralized architecture [CPW89], [Mars96]: It has a single manager that handles all the agents and the application processes, makes decisions and initiates reconfiguration.
- distributed architecture [LF95]: It has multiple management domains with a manager for each one. When information from another domain is needed, the manager communicates with its peer systems to retrieve it.

- hierarchical architecture [Herm90], [LF95]: It relies on multiple domain managers and introduces the concept of the manager of managers (MOM). Each domain manager is connected to the rest of the network only through the MOM. The MOM operates on a higher hierarchical level, retrieving information from and coordinating the domain managers. Unlike the distributed architecture, the domain managers do not communicate directly.
- general “network” architecture [Herm90]: It is a combination of the distributed and the hierarchical architectures.

2.4 Dependability Modelling Techniques

The purpose of dependability modelling and evaluation is to gain insight into systems that have not been built, or that are running under conditions that are not available for measurements, or that fail very infrequently.

There are two main classes of techniques for solving models: analytical and simulative techniques. *Analytical* techniques impose restrictions on models in order to ensure existence and feasibility of analytical solutions. These techniques can be classified as: *closed-form* (where a measure of interest is given as an explicit expression in terms of model-structure and parameters) and *numerical* (where a measure of interest is found by solving a system of equations using numerical analysis techniques, e.g. iterative procedures). *Simulative* techniques, on the other hand, mimic the system behavior by executing an appropriate simulation program and deriving statistical estimates of the measures of interest. Although analytical techniques consider a restricted set of models, they can provide a good insight into the aspects that affect the system dependability at a lower cost than simulation.

This research focuses on analytical techniques. Simulative techniques are not surveyed here. The naive simulation techniques are not efficient since the events of interest (e.g. a system failure event) are rare. The interested reader is referred to Nakayama [Naka94], Heidelberger [Heid95] and Nicola, Shahabuddin and Nakayama [NSN99] for comprehen-

sive surveys of fast simulation techniques for reliability models, and to Heidelberger [Heid95] and Asmussen and Rubinstein [AR95] for techniques applied to queueing and queueing network performance models.

2.4.1 Reliability and Availability Modelling Techniques

There are two main classes of models that are used to compute the reliability and availability of a system: Non-State-Space models and State-Space models. Figure 2.4 shows the model classification. This section briefly describes their modelling capabilities, advantages and disadvantages.

Figure 2.4 Reliability/Availability Models

Non-State-Space models

Series-parallel reliability block diagrams, reliability graphs and fault trees are the three commonly used non-state-space models [STP96]. They are all similar in the sense that they all capture the conditions that make the system fail in terms of the structural relationships among components. They differ in whether the components are represented as nodes or edges in the model, and whether the components can be repeated (appear in more than one place in the model) or not.

In a *series-parallel reliability block diagram* [STP96], the components are represented as nodes and they are combined into blocks in series or parallel or k-out-of-n structure. All of these structures can be there in a single diagram. Each node may have a failure probability or a failure rate, a failure distribution function or unavailability associated with it. Components that appear more than once are assumed to be copies with independent and identical distribution functions. These models can be solved using series-parallel reduction techniques which allow for fast computation. However, not many systems follow such series-parallel structure.

A *reliability graph* [STP96] consists of a set of nodes and edges where the edges represent the components that can fail. Each edge may have a failure probability or a failure rate, a failure distribution function or unavailability associated with it. The graph has a source node (node with no incoming edges) and a sink node (node with no outgoing edges). The system represented by the graph fails if there is no path from source to sink. The model solution in this case requires a factoring algorithm in addition to series-parallel reduction.

A *fault tree* [STP96] represents the combination of component failures that cause the occurrence of system failure in a tree-like structure. It uses boolean gates (AND, OR and k-out-of-n) to represent the combinations. If two gates share an input, then the fault tree is said to have repeated events. Fault trees can be solved using series-parallel reduction (used for the case of without repeated events), factoring [STP96], cutset-based techniques [MW82] or binary-decision-diagram [CM94], [DD95] based approach.

The above three models are examples of *network reliability models* [Colb87]. In network reliability models, we try to find if given two nodes are connected or not, where the edges might fail. These models can be solved using a factoring algorithm, however, a more efficient way is as follows:

- generate all the minpaths or mincuts. A *path* is a set of edges such that when all the edges are up, the given two nodes are connected. Similarly, a *cut* is a set of edges that disconnects the given two nodes. A path is *minimal* if it has no proper sub-paths.

Similarly, a cut is minimal if it has no proper subcuts.

- if all the minpaths (or mincuts) are disjoint, then compute the system reliability as the sum of their reliabilities. Otherwise if they are not disjoint, then apply inclusion-exclusion [Colb87] or sum-of-disjoint products techniques [VT91], [RVT95], [HR87], [Abra79] to find the system reliability.

The above discussion is for models of coherent systems [MW82], in which “a system is in a failed state when all of its components are in failed states; it is in a working state when all of its components are in working states; and, if it is initially in a working state for a state of its components, then it remains in a working state whenever some components which were initially in failed states are restored to working states”.

The coherence property is unsatisfied in some cases. For example, if a fault tree has NOT, NAND or NOR gates, then the system might not be in working state when all of its components are in working states and thus dissatisfies the coherence property. Such a fault tree is called a *non-coherent fault tree*. These fault trees are solved using minimum prime implicants technique (instead of mincuts) [DR97]. Figure 2.5 shows an example of a non-coherent fault tree. This thesis combines the concepts of non-coherent fault trees (to compute the failure propagation through a system) and network reliability models (for representing the management architecture) for availability computations.

Figure 2.5 An example of a non-coherent fault tree with basic events P1-P4

The major advantage of using the non-state-space models described above is that they are efficient to specify and solve. They can be analyzed based on the states of the components and the system structure without having to consider all combinations of which components have failed. However, they assume that the individual components are independent of one another in their failure and repair behavior.

State-Space Models

State-space models such as Markov chains and Petri-net represent the combinations of states (failed or working) of all components. They have the useful ability to model dependencies in the failure and repair behavior of components, for example, dependent failures or components sharing a repair facility, however, this comes at a price of state-space explosion. A system having n components may require 2^n states.

Markov chains [BGMT98] have the notions of *state* of a system and *transitions* between states. The system is said to occupy a certain “state” whenever it satisfies the conditions defined for that state. The dynamical changes in the state of the system are referred to as “state transitions”. These models require the use of Markov property which states that “given the present state of the system, the future behavior of the system is independent of the past behavior”. In Markov model, it is assumed that the sojourn time (the amount of time spent in a state) is exponentially distributed. For dealing with non-exponentially distributed sojourn times, semi-Markov models were developed. The state space for the Markov models grow much faster than the number of components being modelled, making it difficult to specify a model correctly. This difficulty can be overcome by using Petri nets which are described next.

Petri net models [BGMT98] consist of a set of places, a number of tokens that reside at the places and a set of transitions that represent events causing tokens to move from one place to another. These models are analyzed by automatically converting them to Markov models. Although Petri nets are more concise and closer to designer’s intuition than Markov models, but from model solution perspective, they are often the same.

Hybrid models

These models use the state-space methods for those part of the system that require them and use non-state-space methods for the well-behaved parts of the system that follow the independence assumption. For example, dynamic fault trees [DBB92], [RD98] combine fault-trees and Markov chain techniques to model functional dependencies among components.

2.4.2 Use of Architecture in Models

Black-box models and architecture-based models are used to model software failures. In *black-box* models [WP00], the software system is considered as a whole and only its interactions with the outside world are modelled, without looking into its internal structure. The work [HM96] points out that these models are applicable only very late in the software life cycle, they ignore the information about reliabilities of components of the software and they do not take into account the architecture of the software. With the use of object-oriented design, web-based development and component-based development, it is inappropriate to model the overall failure process of software as a whole and as a consequence architecture-based models are needed.

Architecture-based models are surveyed in [GT01] and they can be classified as: state-based or path-based. In *state-based* models, it is assumed that the transfer of control between components follow Markov property and they model software architecture with a discrete-time Markov chain [Cheu80], [GWTH98], continuous-time Markov chain [Lapr84] or semi-Markov processes [Kuba89]. The reliability of the application is then computed either in a monolithic way [Cheu80] by solving the composite model that combines the software architecture with the failure behavior of components or in a hierarchical way [GWTH98], [Lapr84], [Kuba89] by superimposing the failure behavior on the solution of the architectural model. The problem with these models are that, first, they are state-based and suffer from state-explosion problem and second, most of them considered sequential applications and not concurrent ones.

In *path-based* architectural models, the reliability is estimated by considering all the possible execution paths of the program. Shooman [Shoo76] assumes that the frequency with which each path is executed and the failure probability on each execution is known. Krishnamurthy and Mathur [KM97] estimates the path reliabilities experimentally while Yacoub et al. [YCA99] takes an tree-traversal algorithmic approach.

The models described above do consider the software architecture in the analysis, however, they are inadequate for applications that employ a separate management architecture for failure detection and recovery. Sun et al. investigated the impact of a single centralized fault management server on a highly available communication system in [SHL02] but did not include more general management architectures. The author believes that her report [DW02] was the first attempt to model the effect of management subsystem failures on system reliability.

Consideration of a management architecture influences the fault coverage of the system where *fault coverage* measures the ability of the system to recover. This is because the system's knowledge about failures in order to recover from them now depends on the status of management components and also on the management architecture. In reliability modelling, the usual approach to model coverage is to have three states *{not failed, failed covered}* (which implies that the system has automatically detected and recovered from the fault), *failed not covered* (which implies that the global system failure has occurred due to the fault regardless of the state of the application)} for each component and then combine the aspects of behavioral decomposition, sum-of-disjoint products and multi-state solution methods [DDP95], [ADM99]. These models cannot be used for layered software architectures. The reason is that there may be multiple reconfiguration points that must be activated in order to tolerate a single failure. Success at a reconfiguration point depends on the system structure and the connectivity between the source of failure and the point of reconfiguration in the fault management architecture. In fact, a failure may be fully, partially or not covered at all by the system, depending on how many of the necessary reconfigurations are successful. Different degrees of coverage typically result in different operational configurations of the system. When a failure is partially covered, the system may end up

failed or may operate with degraded performance, compared to the fully covered case. The present research captures this effect of partial coverage into the dependability analysis.

2.5 Performability Modelling Techniques

Why performability modelling is important

Dependable fault-tolerant computer systems are designed to continue operation despite the presence of component or subsystem failures, although with degraded performance. For example in aircraft flight-control systems, in order to avoid catastrophic failures, it is essential that they meet hard real-time constraints. System outage for even a short period of time cannot be tolerated. Thus, they are designed with sufficient redundancy so as to ensure that the correct and timely operation of the system is met even in the presence of failures of certain components. On the other hand, in transaction processing systems like banking systems, it is important that certain response time requirements are satisfied, but minor interruptions in service are tolerable. Pure performance analysis of these fault-tolerant systems, which considers resource contention but ignores the failure and repair behavior, generally overestimates the system's ability to perform a certain job. In contrast, pure reliability/availability analysis of these systems tend to be too conservative since the performance considerations are not taken into account. As a consequence, for proper modelling of these systems, it is essential to develop measures and modelling techniques that combines the performance and the failure-repair aspects, popularly known as *performability* [Meye80]. This thesis uses measures of performability.

There are two types of models that are used to compute the performability of a system: Monolithic models and Hierarchical models. Figure 2.6 shows the model classification. This section briefly describes their modelling capabilities, advantages and disadvantages.

2.5.1 Monolithic Models

The monolithic approach [TMWH92] is to combine the performance and availability behavior into a single model. It uses state-based models like Markov chains and Petri nets.

Figure 2.6 Performability Models

The advantage of this approach is that it yields accurate results. However, the gain in accuracy can be negated due to two problems: *largeness* and *stiffness*. The state-space of this model is approximately the cross-product of the state-spaces of the availability and performance models (*largeness*). In addition, this model is stiff because the performance related rates (such as job arrival rates) are much larger than the failure and repair related rates. The largeness problem can be handled in two ways: largeness tolerance and largeness avoidance. The largeness can be tolerated by using automatic generation methods for Markov chains. Haverkort and Trivedi have summarized these approaches in [HT93]. These methods address only the issue of model specification and generation but not the model solution (the underlying CTMC is still large). The largeness problem can be avoided by utilizing different kinds of approximations, such as, truncation, state lumping

and model composition which have been explained in [TCMS93] (a hierarchical model composition technique is described in next subsection 2.5.2). Stiffness problems can be handled by applying aggregation techniques [BT86] and stiffness-tolerant [MMT94] methods.

2.5.2 Hierarchical Models

The hierarchical approach, taken in the present research, is to compose the overall model of the system from a set of smaller non-stiff models. It avoids both the largeness and the stiffness problems of monolithic approach.

This approach is based on the following assumption. Since the failure and repair related rates are several orders of magnitude smaller than the performance related rates, we can assume that the system attains a steady-state with respect to performance related events between successive occurrence of failure-repair events, i.e. the performance measures would reach a stationary condition between changes in the system structure. The performance measures can be computed for the system in each of these steady states. The overall system can then be characterized by weighing steady-state performance measures by structure state probabilities. This leads to a natural hierarchy of models: a higher-level availability model (also called *structure state* model) representing the failure and repair behavior of components and a set of lower-level performance models (also called *reward* models), one for each structural state in the availability model.

Several authors have used this approach for performability analysis. Beaudry [Beau78] computed the computational availability until failure and Meyer [Meye80] proposed the framework of performability for fault-tolerant systems. Ma [MHT01] compared this approach with the monolithic approach for a M/M/C/C queueing system in a wireless communication network. Various tools that automate the approach are surveyed in detail by Haverkort et al. in [HMRT01], [HN96].

Hierarchical models are either *state-space-based* or *non-state-space-based*.

State-space-based: Markov Reward Model (MRM)

An MRM consists of a Continuous Time Markov Chain (CTMC), $X = \{X(t), t \geq 0\}$ with a state space S and a reward function assigned to its states as well as to its transitions [Howa71]. In the former case, the model is called *rate-based* MRM; in the latter case the model is called *impulse-based* MRM. Combination of two types are also possible. [Meye80] adopted the MRMs for performability analysis and for this purpose, usually rate-based MRMs are considered so those are discussed here. Let us now define a reward rate vector $\mathbf{r}$ over the states of the CTMC such that a reward rate r_i is assigned to state i . Let $Z(t)$ denote the instantaneous reward rate of MRM, then $Z(t) = r_{X(t)}$.

Let π_i denote the steady-state probability of residing in state i and $p_i(t)$ the transient probability of residing in i at time t . The following basic measures can be distinguished:

1. The expected instantaneous reward rate $E[Z(t)]$ or point performability (PP) is given by:

$$PP(t) = E[Z(t)] = \sum_{i \in S} r_i p_i(t)$$

2. The expected steady-state reward rate, $E[Z]$ or steady-state performability (SSP) is defined as:

$$SSP = E[Z] = E[Z(\infty)] = \sum_{i \in S} r_i \pi_i$$

3. Let $Y(t)$ denote the accumulated reward in the interval $[0, t)$. Then, $Y(t)$, also known as cumulative performability (CP) is given by:

$$CP(t) = Y(t) = \int_0^t Z(\tau) d\tau = \int_0^t r_{X(\tau)} d\tau ; Y(t) \text{ is a random variable.}$$

4. The probability that the instantaneous reward rate $Z(t)$ does not exceed a certain level z is given as: $P(Z(t) \leq z) = \sum_{i \in S, r_i \leq z} p_i(t)$

5. The probability that the steady-state reward rate Z does not exceed a certain level z is given as: $P(Z \leq z) = \sum_{i \in S, r_i \leq z} \pi_i$

In MRM, no assumptions are made about how the reward rates are obtained. They

may be obtained by simulation, by solving a queueing network or solving other performance models. They are assigned measures such as expected throughput, job loss probability and so on [MT91], [MT92].

There are various higher-level modelling formalisms based on MRM approach. For example, Stochastic Reward Nets (SRN), Stochastic Activity Networks (SAN) and System Availability Estimator (SAVE) models. A SRN model [CBCMT93] extends a Generalized Stochastic Petri Net (GSPN) by allowing reward rates to be associated to the markings. The reward rates can be specified as C functions and they can invoke some other functions to do some performance analysis. A SRN model can be solved by SPNP tool [CMT89] which automatically converts it to an MRM.

SANs [MMS85] are extensions to GSPNs with primitives such as activities (same as “transitions” in Petri nets), places, input gates and output gates. The input and output gates are used to model the enabling and completion of activities. In a SAN, both impulse-based and rate-based rewards can be specified. Impulse rewards are assigned to activity completions and rate rewards are assigned to markings. These models can be solved by the UltraSAN tool [CFJOQRST91], [SOQW95]. For solution, the tool converts a SAN into a state-level representation. If the model is Markov in nature, then a Markov model is generated. The tool supports both analytical and simulation techniques. The rewards are specified as general C functions of the SAN markings and/or activity completions. Although the reward rates are not derived from a performance model but they can be an implementation of some queueing model.

The SAVE modelling language [GCSLT86] consists of a set of modelling constructs for describing the components in a system, their failure and repair characteristics, the dependencies between components, and the conditions on the individual components for the system to be considered available. These models can be solved by the tool SAVE [GCSLT86] which parses an input file and creates an MRM. In this case, the reward rates have to be supplied by the user.

In all the above three modelling formalisms, mostly the reward rates are not derived

from a performance model.

The *Dynamic Queueing Network* (DQN) approach given by Haverkort and Niemegeers [HN89], [HN91], [HNZ92] is another modelling formalism based on MRMs where the performance and dependability aspects of a system and their interdependence are formally specified. Here, a parameterized queueing network is used to model the performance aspects of the system and a GSPN is used to describe the availability aspects of the system. This technique uses a function Φ that maps the possible markings of the GSPN, denoting the system configurations, to the queueing network parameters. Then, a reward rate is associated with every possible GSPN marking by solving the corresponding queueing network models. Finally, the actual calculations for performability are done by translating the DQN model to an MRM. This approach has been automated by a tool called DyQNTOL+ [Have95]. The DQN approach has considered the problem of minimizing the number of performance solutions required. It groups the markings corresponding to the same queueing network parameters and solves the queueing network model once for each group. The DQN approach is similar to the work presented in this thesis in the way that both have considered the problem of minimizing the number of performance solutions required. However the work in this thesis takes advantage of the layered queueing network concept [Wood89], [WNPM95], [RS95] which can model the logical resources and treats the software and hardware contention uniformly, unlike the queueing network concept.

MRM approaches suffer from the state-space explosion problem. In a MRM, since a reward rate is attached to the various states of the Markov model, it is possible to partition the state space into equivalence classes, each class being represented by an aggregated state which is defined by the set of states that have the same performance model and thus have the same value of the reward rate (i.e. the same performance model). For efficiency in the performability calculation, it is therefore necessary to devise a method that would determine these aggregated states, thereby minimizing the number of performance solutions required and compute the probabilities of these states without solving the large Markov model and without generation/enumeration of all the states of the system.

Performance Modelling as part of performability

There are several models suitable for performance analysis: Queueing Networks, Markov chains, Petri nets, Layered Queueing Networks.

A Queueing Network (QN) [BGMT98] model is a collection of service centers (each with its own queue and servers) and jobs (requesters of service). Every service center represents a common resource, shared by jobs, and has a scheduling discipline to dispatch the requests. After completion of service at one service center, a job may proceed to another service center according to a routing chain. They have highly efficient solution methods. Although they were designed to represent competition for limited resources, realistic situations such as concurrency within a job, synchronization and simultaneous resource possession violate the assumptions required for the efficient solution.

Markov chains and Petri nets [BGMT98] overcome the limitations of QNs. They provide a general framework that can be used to model all the above mentioned properties of systems. Petri nets are a higher-level specification of a Markov chain and are more concise. Unfortunately, both of them suffer from the large state space problem.

A Layered Queueing Network (LQN) [Wood89], [WNPM95], [RS95] is an extension to the QN model where the main difference is that in a LQN, a server may become a client to other servers, thus giving rise to nested services. This model is a framework for layered distributed systems that use synchronous inter-process communication, where the sender is blocked until the receiver replies. It can also accommodate asynchronous interactions (with no reply). It can be solved (analytically or by simulation) for various performance measures using the LQNS tool [FMNPRW96]. Figure 2.7 illustrates an example of an LQN model. There are six *tasks* (concurrent operating system processes, represented as rectangles), “Console”, “Application Server”, “Console Server”, “Log Server”, “Database-A” and “Data Server”. Each task runs on a processor represented by an ellipse. Each task and each processor has its own message queue. The task “Application Server” creates a report which involves reading from the database, requesting some kind of application data from the “Data Server” and then logging the report to the “Log Server”. Tasks have

Figure 2.7 An example LQN model

one or more entries which are service handlers embedded in them (task “Data Server” has entries “Get Application Data” and “Get Console Data”). The workload is generated by tasks like “Console” which cycle and create requests, and they are called *reference tasks*. The entries of reference tasks are called *reference entries*. The model is restricted to being acyclic in order to avoid cycles of mutual waiting that may lead to deadlock. The present research uses the LQN model to compute the reward rates for the system operational states.

Extended Markov model

There are two ways to handle models that do not follow the simple Markovian property (i.e. the sojourn time in any state is exponentially distributed and every state transition in a CTMC acts as a regeneration point for the process).

One way is to approximate it by phase type distributions, also called phase approximation as described in [TCMS93]. Although phase approximations enables the use of CTMC for modelling of non-exponential distributions, its major drawback is that it normally results in a large state space.

Another way is to use semi-Markov Reward models [CMST92] (which allows the sojourn time to have any distribution) or Markov regenerative Reward models [LTP95] (where all state changes do not have to be regeneration points). These variations are not surveyed here.

Non-state-space-based: Fault Tolerant Layered Queueing Network (FTLQN) model

This approach has been developed by Das and Woodside [DW98a], [DW98b], [DW01a] for analyzing the class of systems that has layered or tiered software architecture. It uses AND-OR-like graphs for describing the availability aspects of the system and LQN concept for modelling the performance aspects of the system. The model solution can be obtained without solving large Markov models. This model forms the basis of this thesis and is further described in Chapter 3.

Unlike MRM, this model partitions the state space into classes, each class being represented by an aggregated state which is defined by the states that have the same performance model, and computes the probabilities of those classes without generating the whole state space of the system.

Chapter 3

The Dependable-LQN Model

This chapter introduces a formal mathematical representation of an efficient performability model, termed as the *Dependable-LQN* model, for fault-tolerant layered distributed systems. The design of the system may take advantage of passive or active replication to achieve fault tolerance and it may employ a separate management system for carrying out the failure detection and reconfiguration functionalities. A particular strength of this model is that it can evaluate large systems.

Section 3.1 defines the model. Section 3.2 describes the failure propagation in layered systems and Section 3.3 describes the propagation of knowledge about a failure or a repair event in a fault management system. Section 3.4 describes the measures that can be obtained from the model.

3.1 The Dependable-LQN Model Definition

A Dependable-LQN model combines two parts:

- Fault-Tolerant Layered Queueing Network (FTLQN) submodel for describing the layered software architecture
- Model for Availability Management Architecture (MAMA) submodel for describing the fault management architecture

This section first describes the assumed failure behavior for system components and then it describes the submodels, FTLQN and MAMA, that comprises the Dependable-LQN model.

3.1.1 Failure behavior of a single independent component

Individual components may fail independently, or due to dependence on other factors. Those components which fail and are repaired independently, are governed by the Markov

chain in Figure 3.1. It has two states, working (state 1) and failed (state 0). The failure rate is λ and the repair rate is μ .

Figure 3.1 Markov model for each component

Let o_i denote the state of component i , with $o_i = 1$ if component i is in state 1

$= 0$ if component i is in state 0.

The above Markov chain can be solved to obtain the steady-state probabilities, π_0 and π_1 , and the transient-state probabilities (using the initial condition that the component is in state 1 at time $t = 0$), $p_0(t)$ and $p_1(t)$ as:

$$\pi_0 = \lambda / (\mu + \lambda)$$

$$\pi_1 = \mu / (\mu + \lambda).$$

and

$$p_0(t) = (\lambda / (\mu + \lambda)) - (\lambda e^{-(\mu + \lambda)t} / (\mu + \lambda))$$

$$p_1(t) = (\mu / (\mu + \lambda)) + (\lambda e^{-(\mu + \lambda)t} / (\mu + \lambda))$$

Services offered by these components may fail due to service dependencies, even though the component has not failed. These dependencies are described by FTLQN model, which is described next.

3.1.2 First part of Dependable-LQN model: FTLQN model

An FTLQN model is introduced in [DW01a] and extended here for additional policies for replication and extra failure dependencies that cannot be represented by service-based dependencies. It extends the Layered Queueing Network (LQN) performance model by specifying the strategies to be used in case of failures and by adding redundant components and availability related parameters to it.

Figure 3.2 shows the graphical notations used for FTLQN model. The notations will be defined in more detail in the relevant context in this subsection.

Figure 3.2 FTLQN notations

Figure 3.3 shows an example of an FTLQN model of a layered console application. This model extends the LQN model shown in Figure 2.7 of Chapter 2 by adding the redundant task “Database-B” (backup of “Database-A”) that runs on “Processor 3”. In this model, a *service request* (represented by a rounded rectangle) has a name. Further, the service request has a single target server in case there is no server redundancy, or it may have a set of redundant target servers with a policy on how to use them.

The different redundancy policies supported for a *service request* are:

- Primary-Standby Redundancy (PSR) with load sent only to the primary target. When primary fails, one of the standbys will become the primary and the load would be rerouted to the new primary.

Figure 3.3 An FTLQN model

- Active Redundancy (AR) with load replicated and sent to all the working target servers.
- Primary-Standby Active Redundancy (PSAR) with load replicated and sent to all the working target servers where a designated primary does exist. This redundancy policy is useful in cases where a client sends a service request to all the redundant servers (in Layer i) for close synchronization, however only the primary server of Layer i would send the service request to Layer $i+1$ in order to decrease the number of replicated requests to Layer $i+1$.
- Load-Balanced Redundancy (LBR) with load equally divided among all the working target servers.

In Figure 3.3, “dbService” is a service requested by the entry “Create Report” for read-

ing from the database. It has PSR policy where the selection sequence of the target servers is labelled by “#n” on the arcs going out from the service to the server(s). In this figure, all the service requests are synchronous except “serv5” which is asynchronous. Asynchronous requests are assumed to be non-essential to the entry which makes them, so they do not have retargettable services. This can be generalized, and a failure dependency can be modelled. In order to add additional failure dependencies that cannot be represented by service-based dependencies, another abstraction called *depends* relationship can exist between entries. The *depends* relationship may exist from an entry to a single entry, or to multiple entries with “AND” or “OR” types. For example, the entry “CreateReport” fails if the logging mechanism fails even though the communication is asynchronous. This would be a depends relationship of “CreateReport” on the entry “LogReport”. “AND” and “OR” types have been used in the case study in Chapter 6.

The performance parameters for the FTLQN model are the mean total demand for execution on the processor by each entry and the mean number of requests for each interaction. The availability related parameters for this model are the failure rate and repair rate for each component (either a task or a processor).

The special property of layered systems is that a failure of a task or processor in one layer can cause many tasks that depend on its services (at any layer in the system) to fail, unless there is a redundant server. In general, there can be any number of layers in the system, and network components can be included. This model captured the layered operational and failure dependencies, and this could be generalized to include abstract “failure dependency factors” that model some forms of dependency among individual failures, which is described next.

Dependent failures

The failures of different services are dependent because of the way they share lower-level services of the system, and this is all captured by the layered structure of the model. However, so far it has been assumed that the processors and the software tasks undergo independent failure. If this is not the case, the model can be extended in a straightforward way

to model the dependencies.

For hardware components, consider two computers in the same room. Each one has independent sources of failure within its own box, such as power supply, memory, connectors. There are also shared influences, such as failure of power to the room, or vibration in the room that could cause a mechanical failure in both computers. This can be modelled using a *depends* relationship between the two computers and the separate and shared failure factors. In Figure 3.4 these “failure factors” are introduced as additional nodes, called *pseudocomponents*, in an FTLQN model. For example, processor P1 is up if both of its “separate factor” AND the “shared factor” are up. These *depends* relationships between processors and pseudocomponents do not indicate performance-related dependencies, but just indicate failure dependencies. These additional pseudocomponents influence failures through AND-OR relationships, and these new leaf nodes are assumed to have independent failures.

Figure 3.4 Modelling dependent failures with AND-OR depends relationships

For software tasks, failure dependence between services is already modelled by shared service dependencies. However, there can be other dependencies which can be modelled by shared factors. This could be because the model is too high level to show the dependence. For instance, two of the tasks on a processor might share a buffer, and a bug in the

buffer handling might crash them both. The shared buffer would be a shared factor, which would not affect other tasks on the same processor. A simple way to extend the model to shared factors between tasks is to add a pseudocomponent, as shown in Figure 3.5.

Figure 3.5 A shared software factor introduced as a pseudocomponent

The failure of an entire processor node can be due to a hardware failure or to an operating system failure. It is possible to regard these both as failures of the processor, since they have the same effect of failing all tasks on the node, but there are important differences. In particular, an operating system failure may be repaired by rebooting the system, while a hardware failure requires a human repair operation. Thus their repair rates will be very different. To separate these out, (and assuming that the operating system is not modelled in detail in the layered model, but rather its workload is combined with that of the tasks on the node), an “operating system factor” and a “hardware factor” can be introduced as failure factors to the processor entity. This is illustrated in Figure 3.6.

This approach to modelling failure dependencies is only limited by one’s capability to model dependency among failures by AND-OR relationships.

Operational Configurations

In reaction to a failure, the fault management subsystem reconfigures the system. It can respond not only to processor failures, but also to software failures (task crashes or operating system crashes). This resolution of requests gives different *operational configurations* of the system. For example, in Figure 3.3 if all the tasks are operational, then the configu-

Figure 3.6 Separating the failure influences of the operating system and the hardware for a processor. The two factors (shown as pseudocomponents) have different failure and repair rates

ration is the system as shown, but with Database-B, and its service requests (labelled #2) removed as they are not used. The probabilities of the operational configurations depend on the fault management architecture, management subsystem failures, and as well as on the application.

The next subsection describes the management components, their connections and how they are related to the application components.

3.1.3 Second part of Dependable-LQN model: MAMA model

The MAMA model describes fault management architectures based on the manager-agent model described in subsection 2.3.1. Figure 3.7 illustrates the classes involved using UML class diagram notation [BRJ98]. The MAMA notation described next is used to define architecture based on these classes.

The MAMA model has five types of components: *application tasks* (which may include subagent modules), *agent tasks*, *manager tasks*, *pseudocomponents* and the *processors*. Components may be simple or composite, where a composite component contains other components. A processor is automatically a composite component that contains a cluster of tasks that execute there. If the processor fails, all its enclosed tasks fail. A

Figure 3.7 The manager-agent model

pseudocomponent may be a composite component or a non-composite one. These components are connected using three different classes of connectors:

- *Alive-watch connector*: This connector is used between two components in cases where the destination component would like to be able to detect whether the source component is alive or not. This may be achieved by periodic polls or “i-am-alive” messages. Usually, the source of this connector are the manageable application components.
- *Status-watch connector*: This connector is used in cases where a destination component would like to know about the liveness of the source component and also wants to collect failure data about other components in the system that have been gathered by the source component. An example would be a connector from an agent task to a manager task.
- *Notify connector*: This connector is used in cases where the source component would like to send or forward reconfiguration commands to its sub-ordinate destination

component (for example, a manager sending commands to an agent or an agent forwarding a command to an application task) or conveying management information to its peer (for example, a manager sending information about the domain it manages to another manager).

These connectors are typed according to the information they convey, in a way which supports the analysis of knowledge of the system status at different points in the management system.

Upon occurrence of a failure or repair of a task or a processor, the occurrence is first captured via alive-watch or status-watch connections and the information propagates through status-watch and notify connections, to managers which initiate system reconfiguration. Reconfiguration commands are sent by *notify* connections.

The further discussion makes two assumptions. First, cycles may occur in a MAMA model, but we assume that the information flow is managed so as to not cycle. Second, we assume that if a task watches a remote component, then it also watches the components containing the remote component, in order to distinguish between the composite component failure and the non-composite failure.

Figure 3.8 shows the graphical notation used for MAMA components and connectors.

Figure 3.9 shows a MAMA model for a centralized management architecture for the application of Figure 3.3, with agents ag1 to ag4 running on the processors Processor1 to Processor4. m1 is the single manager task that indirectly manages all the application tasks through the agents and directly monitors the processors.

For MAMA model, the failure rate and repair rate can be provided for all management components and the connectors between them.

The next section describes how the failure information propagates in the management architecture from the source of failure to the point where reconfigurations are executed.

Figure 3.8 MAMA notations

3.2 Modelling Knowledge Propagation in MAMA

This section describes how the MAMA model affects the system recovery by limiting the knowledge about the failure states. It is assumed that if the MAMA manager/agent connected to a component that must reconfigure has knowledge of the failure, then the reconfiguration will take place.

The responsibility of the fault management system, represented by a MAMA model, is to detect component failures, reason about the global system state, and to issue reconfiguration commands to the relevant retargettable application tasks. Successful reconfiguration depends on the connectivity (through the MAMA model) between the source of failure and the point of retargetting services, at the instant of failure. The connectivity is determined by applying minpath algorithms to a derived graph termed as the *Knowledge Propagation Graph* for the MAMA model.

There are four types of arcs in the knowledge propagation graph K : $\{component, alive-watch, status-watch, notify\}$. Each component in the MAMA model leads to an arc of type

Figure 3.9 MAMA model for a centralized management architecture for layered application of Figure 3.3

component; each connector in the MAMA model leads to an arc of the same type as the connector. A component or connector failure is represented by an arc failure.

Let us denote iv_i and tv_i to be the initial and terminal vertices respectively of arc i .

The Knowledge Propagation Graph, K , is created from the MAMA model as follows.

For each component i in the MAMA model, K has

- two vertices iv_i, tv_i
- a directed arc $i = (iv_i, tv_i)$, of type *component*.

For each connector (i, j) in the MAMA model, K has

- a directed arc w from tv_i to iv_j .
- the type of the arc w is set equal to the type of the connector (i, j) . That is, it is one of $\{alive-watch, status-watch, notify\}$.

Figure 3.10 shows the knowledge propagation graph corresponding to the MAMA model in Figure 3.9.

The knowledge propagation graph is used for computing the binary-valued connectivity function, $know_{c,t}$, between two components c and t as follows:

$$know_{c,t} = \bigvee_{q=1}^m \left(\bigwedge_{j \in P_q^+} o_j \right)$$

where o_i represents the state of component i and m is the total number of minpaths ($P_1, P_2, \dots, P_m$) from tv_c to tv_t . A minpath P_q is a minimal set of arcs of graph K such that when all the arcs in P_q are operational, then vertices tv_c and tv_t are connected; vertices tv_c and tv_t are disconnected for every proper subset of P_q . A minpath P_q from tv_c to tv_t is obtained from K when c represents a task, or from the reduced graph $[K - \{\text{arcs representing task } t_j \text{ (contained in processor } c) \text{ in } K\}]$ when c represents a processor. The analysis can use any standard minpaths algorithm (e.g. [Colb87]), taking into account that the first arc in the path must be of type *alive-watch* or *status-watch* and rest of the arcs should be of type *component*, *status-watch* or *notify*. Define P_q^+ as an augmented minpath obtained from P_q as:

$$P_q^+ = P_q \cup \left\{ \bigcup_{t_j \in P_q} (\text{arc } p \mid p \text{ is processor of task } t_j) \right\} \cup (\text{arc } p_c \mid p_c \text{ is processor of task } c) \quad \text{if } c \text{ is a task, or}$$

$$P_q^+ = P_q \cup \left\{ \bigcup_{t_j \in P_q} (\text{arc } p \mid p \text{ is processor of task } t_j) \right\} \quad \text{if } c \text{ is a processor.}$$

Each edge corresponding to a component is labelled by its name and type as name; type.
 Abbreviations are: *cmpt* = component; *ntfy* = notify;
aw = alive-watch; *sw* = status-watch

Figure 3.10 The Knowledge Propagation graph corresponding to the MAMA model shown in Figure 3.9

The knowledge propagation graph is used in the next section while modelling the operational dependencies of the FTLQN model.

3.3 Modelling Fault Propagation

The operational dependencies describe how an operation depends on its processor and on layered services, and on the use of a “working” alternative for a failed service. Earlier analysis, based on perfect knowledge of failures, used an AND-OR graph to model the dependencies [DW01a]. In the present model a “working” state may depend (through reconfiguration) on the fact that some system element is in a “not working” state (e.g. a backup configuration of an application might corresponds to a primary element being “not working” and the management system has knowledge about the failure and has reconfigured the application to use the backup). This requires an AND-OR-NOT graph, like the *Non-coherent fault-propagation graph* introduced here and also described in [DW04a]. It represents the operational dependencies among the entries in the FTLQN model and also takes into account the effect of failures of the management and the agent tasks on the operation of the entries. It combines the management connectivity informations (computed from the knowledge propagation graph) with the structural information extracted from the FTLQN model.

3.3.1 Fault Propagation Graph Definition

The fault propagation graph, G , is an AND-OR-NOT graph that has a root node *root*, and a node for each processor, task, pseudocomponent, entry, and service. Some dummy nodes are also added to this graph to account for the knowledge information. Before going to the graph definition, let us define the following terms:

- $know_{c,t}$ as defined in Section 3.2 is a boolean function of the system state which takes the value 1 if task t has knowledge about the status of a component c , as determined by the connectivity of the fault management architecture. Otherwise, it takes the value 0.
- $know_{e,t}$ is a boolean function which takes the value 1 if entry e is in working state and task t knows this through its knowledge (via the fault management architecture) of the state of components that affect e . It can be determined from the FTLQN model

and $know_{c,t}$, $know_{e',t}$ is the corresponding function for knowledge of the failed state of entry e . Otherwise, it takes the value 0.

The fault propagation graph, G , is a six-tuple $G = (V, E, L, policy, value, prob)$ where

- 1) V is the set of nodes in the graph G , representing elements of the system and sometimes also knowledge information. The various node types are described next in the Node Types subsection.
- 2) E is a subset of $V \times V$, the set of directed edges. Let $child(v)$ denotes the children of node v , i.e. $child(v)$ is the set $\{w \mid w = tv_e, \text{ where the edge } e = (v, w) \text{ and } tv_e \text{ means the terminal vertex of edge } e\}$.
- 3) L is the set of leaf nodes, which can fail independently. For a leaf node l , $child(l) = \emptyset$.
- 4) $policy$ is a function that labels the redundancy policy which is either $\{PSR, AR, PSAR, LBR\}$ to each service node, where service nodes are described in Node Types subsection.
- 5) $value$ is a boolean function $value: V \rightarrow \{0, 1\}$. The value (0 or 1) represents whether the corresponding element is working or not (including its dependencies), and sometimes also represents knowledge of its status as well. For a non-leaf node, the value depends on the value of its child nodes and the relationship with them (AND, OR, NOT). For example, for an AND node v , $value(v)$ is 1 if the values of all its children are 1. For a leaf node l , $value(l) = o_l$, where o_l denotes the state of the component represented by node l .
- 6) $prob$ is a function $prob: L \rightarrow \{r \mid (r \in \mathbb{R}) \wedge (0 \leq r \leq 1)\}$ that assigns a real number $prob(l)$ to each leaf node $l \in L$, representing the probability of node l being in the “working” state at steady-state, i.e. $prob(l) = \pi_1$, or at time t , i.e. $prob(l) = p_1(t)$, depending on the type of measure being evaluated.

In displaying the graph G , edges connecting a node to a group of child nodes represent

an OR function of the values at the lower nodes. If the edges have an arc across them, they represent an AND function. A filled node connecting to one lower node is a NOT function of that node. Leaf nodes represent system components or pseudocomponents which have independent failure behavior.

Node Types

V contains eight types of nodes, namely, *processor* node, *task* node, *pseudocomponent* node, *entry* node, *dummy* node, *service* node, *root* node and *user-defined* node which are defined as follows:

- (i) a *processor* node represents a processor p of the FTLQN or MAMA model. If processor p has independent failure and repair behavior, then the corresponding processor node is a leaf node. Otherwise, its type is chosen based on the kind of *depends* relationship specified for the processor in the FTLQN model on other pseudocomponents. The set of processor nodes is denoted by $V_{processor}$
- (ii) a *task* node represents a task t of the FTLQN or MAMA model. If task t has independent failure and repair behavior, then the corresponding task node is a leaf node. Otherwise, its type is chosen based on the kind of *depends* relationship specified for the task in the FTLQN model on other pseudocomponents. The set of task nodes is denoted by V_{task} .
- (iii) a *pseudocomponent* node represents a pseudocomponent pc of the FTLQN model. If pseudocomponent pc has independent failure and repair behavior, then the corresponding node is a leaf node. Otherwise, its type is chosen based on the kind of *depends* relationship specified for the pseudocomponent in the FTLQN model on other pseudocomponents. The set of pseudocomponent nodes is denoted by $V_{pseudocomponent}$.
- (iv) an *entry* node is an AND node representing an entry e of the FTLQN model. The set of entry nodes is denoted by V_{entry} .

Informally, an entry e is working if its

- task is working
- AND its processor is working
- AND all the services it uses are working.
- AND all the other entries it depends on by *depends* relationships are working.

This implies that the *value* of an entry node representing entry e is 1 if the *values* of e 's processor, e 's task, the services that e uses and the entries on which e depends on by *depends* relationship are 1. Thus the children of the entry node representing entry e are the nodes representing e 's task, e 's processor, entries on which e depends on using *depends* relationship and the services that e uses.

- (v) a *dummy* node can be an AND node or an OR node. They are added to consider the knowledge information. The set of *dummy* nodes is denoted by V_{dummy}
- (vi) a *service* node is an OR node representing a service s of the FTLQN model. The set of *service* nodes is denoted by $V_{service}$. The child nodes of a service node are either entry nodes or dummy nodes. If the service s has a single target entry, then the node representing that entry is its only child. Otherwise, if s has multiple redundant target entries $e_1, e_2, \dots, e_M$, then the children of the corresponding service node are dummy nodes, $d_1, d_2, \dots, d_M$ (say) which represent the operational and knowledge information about the target entries.

To understand the significance of the dummy nodes, suppose that the name of the task that requires service s is t , that the service s has a redundancy policy PSR, and that the target entries, $e_1, e_2, \dots, e_M$ are ordered by their selection priorities 1, 2, ..., M . Then:

- d_1 would be an AND node with two children: an entry node corresponding to entry e_1 and a dummy node, say d_{11} , whose *value* represents the knowledge of *working* status of e_1 by the task t , i.e. $value(d_{11}) = know_{e_1, t}$.

- d_i , where $i = 2, \dots, M$ would be an AND node with the entry node e_i and i dummy nodes, $d_{i1}, \dots, d_{ii}$, as children. The values of $(i - 1)$ dummy nodes represent the knowledge of *failed* status of entries e_j by the task t (i.e. $value(d_{ij}) = know_{e_j, t}$), where $j = 1, 2, \dots, i - 1$, and the value of one dummy node represents the knowledge of *working* status of entry e_i by task t (i.e. $value(d_{ii}) = know_{e_i, t}$).

On the other hand, if the service s has a redundancy policy AR, LBR or PSAR then:

- d_i , where $i = 1, \dots, M$ would be an AND node with two children: an entry node corresponding to entry e_i and a dummy node, say d_{i1} , whose value represents the knowledge of *working* status of e_i by the task t , i.e. $value(d_{i1}) = know_{e_i, t}$.

The dummy nodes whose values represent $know_{e, t}$ are expanded further according to the following logic: In order to know that the entry e is *working*, task t has to know that the processor of e , the task of e , all the services that e uses, and the entries that e depends on are working. These nodes depend, through recursive relationships, on dummy nodes whose values represent the knowledge of task t about the status of all the components c that affect entry e , i.e. $know_{c, t}$. These dummy nodes represent alternative minpaths between two components and thus is an OR node with each child, which is an AND-node, representing a single minpath.

The dummy nodes whose value represents $know_{e, t}$ are expanded further according to the following logic: In order to know that the entry e is *failed*, task t has to know that either the processor of e , or the task of e , or any of the services that e uses, or any (all) of the entries that e OR(AND)-depends on are failed and so on.

- (vii) the root node *root* which is an OR node representing the overall state of the system. Its children are the entry nodes which represent the reference entries of the FTLQN model.
- (viii) an *user-defined* node is an OR or an AND node defined by the user to specify the *depends* relationships.

Figure 3.11 shows the fault propagation graph for the model given in Figure 3.3 and Figure 3.10. In this figure, the entry node “UI” corresponding to the reference entry “UI” is the only child of *root*. The service node “dbService” has two dummy nodes, “d1” and “d2”, as its children. $value(“d1”) = 1$ represents the primary branch is working and $value(“d2”) = 1$ represents the backup branch is working.

The primary branch is working, represented by $value(“d1”) = 1$, if

- the entry “Read-A” is working
- AND the task “Application Server” which requires “dbService” has the knowledge about the working status of “Read-A”. Let the *value* of the dummy node “d3” represents this knowledge information. In order to know that “Read-A” is working (i.e. $value(“d3”) = 1$), the task “Application Server” has to know that the task “Database-A” is working (represented by the $value(“d4”) = 1$) AND also the processor “Processor2” is working (represented by $value(“d5”) = 1$). $value(“d4”) = 1$ requires “Database-A” to be working AND there is a connectivity from “Database-A” to “Application Server”, represented by the value of the dummy node $know_{Database-A, ApplicationServer}$. Node $know_{Database-A, ApplicationServer}$ represents alternative minpaths between two components and thus is an OR node with each child representing a single minpath. For clarity, these dummy nodes are not expanded in this figure.

The backup branch is working, represented by $value(“d2”) = 1$, if

- the entry “Read-B” is working.
- AND the task “Application Server” knows about the working status of “Read-B”. Let the value of dummy node “d6” represents this fact. The expansion of “d6” is very similar to the expansion logic of “d1” and is not elaborated here.
- AND the entry “Read-A” is failed and the task “Application Server” knows about the failure. This fact is represented by the value of the dummy node “d7”. In order to know that “Read-A” is failed (i.e. $value(“d7”) = 1$), the task “Application Server” has

to know that either the task “Database-A” is failed (represented by $value(“d8”) = 1$) OR the processor “Processor2” is failed (represented by $value(“d9”) = 1$). $value(“d8”) = 1$ requires “Database-A” to be failed AND there is a connectivity from “Database-A” to “Application Server”, represented by the value of the dummy node $know_{Database-A, ApplicationServer}$. Similar argument holds for node “d9”.

This graph is used to determine the set of distinct operational configurations of the FTLQN model, and their probabilities. From the root down, all distinct sets of alternative working possibilities are created by a breadth-first search. Once the graph is all searched, a complete alternative corresponds to an operational configuration. A full description of the computations is given in Chapter 4.

3.3.2 Notation for Operational Configurations

Let us consider that there are N leaf nodes which have independent failure behavior and E entry nodes in the Fault Propagation Graph G .

Definition 3.1 The *system configuration state* S is defined to be a vector of values of the leaf nodes, given by $S = \{value(l), l = 1, 2, \dots, N\}$.

Definition 3.2 An *effective node* is defined to be a node whose value is 1 and which is being used for service by the system.

Definition 3.3 An *operational configuration* C is an aggregated state that is represented by a set of effective nodes $\{v\}$. It corresponds to a set of *system configuration states* which give the same set of *effective nodes* from the fault propagation graph and thus have the same performance model and the same reward rate. Each C can therefore be concluded to have a distinct performance model.

Definition 3.4 $g: \{S\} \rightarrow \{C\}$ is a function that maps a *system configuration state* S to an *operational configuration* C as per Definition 3.3. The function g is a surjection since many *system configuration states* may correspond to a single *operational configuration* as per Definition 3.3 above.

Definition 3.5 Define an equivalence relation eq on the set of *system configuration states* as:

For any two *system configuration states* S_1 and S_2 ,

$$S_1 eq S_2 \text{ holds if } g(S_1) = g(S_2).$$

Thus, the set of *system configuration states* can be partitioned by the relation eq into equivalence classes and each class can be represented by a single *operational configuration*. The states contained in an *operational configuration* C correspond to the same performance model.

3.4 Measures of Interest

A Dependable-LQN model can yield the following performability measures:

Let r_i denote the reward rate assigned to operational configuration i . Let π_i denote the steady-state probability of residing in operational configuration i and $p_i(t)$ the transient probability of residing in i at time t . Let $Z(t)$ denote the instantaneous reward rate at time t .

- point performability at time t , $E[Z(t)] = \sum_i r_i p_i(t)$.
- steady-state performability, $E[Z] = \sum_i r_i \pi_i$
- the probability that the expected instantaneous reward rate, $Z(t)$, does not exceed a certain level z which is given by: $P(Z(t) \leq z) = \sum_{i, r_i \leq z} p_i(t)$
- the probability that the steady-state reward rate Z does not exceed a certain level z , which is given by: $P(Z \leq z) = \sum_{i, r_i \leq z} \pi_i$

By choosing a suitable reward rate function, particular measures can be obtained. For example, if the reward rate function corresponds to the throughput of a reference task, then $E[Z(t)]$ would be the expected throughput of that task at time t and $E[Z]$ would be the

expected throughput in steady-state. Using Little's theorem [BGMT98], the mean response time measure can also be computed.

3.5 Summary

This chapter has introduced a formal description of the Dependable-LQN model: a model for evaluating performability of layered systems that employ a separate management architecture for failure detection and recovery. It has presented the relevant definitions that will be used in the subsequent parts of this thesis. It has demonstrated how the model can be transformed into an AND-OR-NOT fault propagation graph exploiting the dependencies of the FTLQN model and the connectivity information from the MAMA model. It has further introduced the measures that can be obtained from the Dependable-LQN model.

Chapter 4

The Dependable-LQN Model Solution

This chapter introduces two different solution strategies, *Enumeration* and *State Aggregation*, for the Dependable-LQN model, described in Chapter 3, in order to obtain the performability measures given in Section 3.4. Both solution strategies use the AND-OR-NOT graph representation of the model for availability calculations and use the Layered Queueing Network analysis for performance calculations. The *Enumeration* strategy involves enumeration of all the system configuration states (the number of system configuration states being equal to the number of states in the underlying Markov Reward model) whereas the *State Aggregation* strategy avoids such enumeration and directly finds the probabilities of the operational configurations.

Section 4.1 describes the solution approaches. Section 4.2 provides some examples to demonstrate the efficiency of the second strategy over the first one for scaled systems. Section 4.3 compares four different fault-management architectures for an application to show the model's capability for comparing different alternatives of management architecture. Section 4.4 evaluates the effect of changing the placement of a fixed number of managers in the management model and shows how the modelling technique can be effective in evaluating those placements.

4.1 The Dependable-LQN Performability Algorithm

This section describes an algorithm to compute the performability measures from Dependable-LQN model.

It has the following steps:

Step(1): Obtain the knowledge propagation graph K corresponding to the specified MAMA model, as described in Section 3.2.

Step(2): Obtain the fault propagation graph G using the layered FTLQN model (i.e. the

layered application) and the knowledge propagation graph K , as described in Section 3.3.

Step(3): Using the fault propagation graph G , determine the set, Y , of all the distinct operational configurations C . A distinct operational configuration corresponds to a distinct choice of alternatives (for a PSR policy) or of different combinations of redundant servers (for a AR or LBR or PSAR policy).

In order to obtain the operational configurations, traverse the graph G in breadth-first-search fashion and during traversal, process a node v as follows:

- if v is the root node, generate $2^{|child(v)|-1}$ operational configurations corresponding to the $2^{|child(v)|-1}$ combinations (excluding all zero combination) of values 0 and 1 for the children of v . In each configuration, add those children nodes whose value is 1 in the corresponding combination.
- if v is an entry node, then add all its children nodes to those operational configurations where v is present.
- if v is a service node, with policy:
 - PSR, then replace each operational configuration C where v is present by $|child(v)|$ number of configurations, C_i , $i = 1, 2, \dots, |child(v)|$, where each C_i consists of the nodes of C and the i -th child of v .
 - LBR or AR or PSAR, then replace each operational configuration C where v is present by $2^{|child(v)|-1}$ number of configurations, C_i , $i = 1, 2, \dots, 2^{|child(v)|-1}$, where each C_i consists of the nodes of C and the nodes whose values are 1 in the i -th combination of values 0 and 1 of the children of v .
- if v is a leaf node (i.e. a processor, task or pseudocomponent node), then no action is taken.
- if v is a *user-defined* node, then no action is taken.

- if v is a dummy AND node, then if it has a child of type entry node, then add all its children nodes to those operational configurations where v is present, otherwise no action is taken.
- if v is a dummy OR node, no action is taken.

Step(4): Compute the steady-state probabilities, π_C , and the probabilities at given time instant t , $p_C(t)$, of the operational configurations C using either the *State Aggregation Approach* or the *Enumeration Approach* as described below.

Step(5): For each $C \in Y$, generate an ordinary Layered Queueing Network model and solve it [FMNPRW96]. From the performance measures, assign a reward r_C to operational configuration C .

Step(6): Compute the steady-state performability of the system as $E[Z] = \sum_{C \in Y} r_C \pi_C$ and the point performability at time t as $E[Z(t)] = \sum_{C \in Y} r_C p_C(t)$. The probabilities $P(Z(t) \leq z) = \sum_{C, r_C \leq z} p_C(t)$ and $P(Z \leq z) = \sum_{C, r_C \leq z} \pi_C$ can also be computed.

Next, we elaborate on Step (4).

Step (4) by State Aggregation Approach:

This approach computes the probabilities of the distinct operational configurations directly without exhaustively generating all the failure states of the system. Using the graph G , it generates a non-coherent fault tree corresponding to each configuration (that has already been obtained in Step(3)) and computes the probability for that configuration using the Aralia tool [DR97]. The fault tree represents a logic function which is true for all states which give the stated configuration, and failed for all other states. Thus, it expresses the conditions under which the configuration occurs. The non-coherent fault tree solution computes the probability of this configuration, from the state probabilities.

The principles of generating a non-coherent fault tree from an operational configuration C is as follows:

- the *top* gate of the fault tree is an AND gate.
- basic events are created corresponding to the leaf nodes in *C* and each of them become a direct child of the *top* gate of the fault tree.
- gates are created corresponding to the dummy nodes in *C* which were not expanded during configuration generation step (Step-3) and they are expanded following the AND-OR logic from the fault propagation graph.
- no explicit gates are created for the entries.

As in our earlier models [DW01a], the representation here is logically inverted compared to most fault trees; an edge is true if the corresponding element is operational (rather than failed). However, the fault tree here is quite different from the earlier models because it represents knowledge dependencies and NOT functions.

In order to demonstrate how a fault tree is created from an operational configuration, let us consider the fault propagation graph (given in Figure 3.11 in Chapter 3) and a configuration containing the nodes {UI, Console, Processor1, serv1, serv2, CreateReport, ApplicationServer, dbService, serv3, LogReport, Fetch, serv4, Processor4, ConsoleServer, d2, d6, d7, Read-B, Database-B, Processor-3, GetApplicationData, DataServer, Processor4, GetConsoleData, LogServer}. This configuration implies “*dbService is using the backup Server Database-B*”. Figure 4.1 provides the non-coherent fault tree generated for this configuration. There are nine leaf nodes in this configuration: “Console”, “Processor1”, “ApplicationServer”, “Processor4”, “ConsoleServer”, “Database-B”, “Processor3”, “DataServer” and “LogServer”. These nine leaf nodes correspond to the nine basic events that are direct children of the top AND gate of the fault tree. No gate is explicitly created for the entries. The dummy nodes “d6” and “d7” are not expanded in the configuration. So there are two gates created for these dummy nodes and they are expanded in the fault tree following the AND-OR-NOT logic of the fault propagation graph in Figure 3.11. In this particular example, all the OR gates corresponding to the know nodes have only one AND child since there is only one minpath between the relevant components.

Figure 4.1 Non-coherent fault-tree corresponding to the operational configuration that implies “dbService is using the Server Database-B” obtained from the fault propagation graph in Figure 3.11 (Chapter 3). Each dotted enclosure realizes a *know* function. The repeated basic events are shown in *italics*

Step (4) by Enumeration Approach:

This brute-force approach enumerates all the failure states. It is described here for comparison.

Let the total number of leaf nodes in the fault propagation graph be N . Enumerate all 2^N system configuration states. For each system configuration state $S = \{value(l), l = 1, 2, \dots, N\}$, where $value(l) = 0$ or 1 , if the *root* node of G is working (i.e. the state S satisfies the fault propagation graph G), obtain the configuration C for that state. Set:

$$Prob(C) = Prob(C) + \prod_{v=1, value(v)=1}^N prob(v) \cdot \prod_{v=1, value(v)=0}^N (1 - prob(v))$$

where, $Prob(C)$ denotes the probability of configuration C and means either π_C or $p_C(t)$ depending on the type of measure being evaluated.

We observe that the complexity of the state aggregation approach grows with the number of operational configurations. The number of operational configurations depends on the structure and the number of components of the FTLQN model. The introduction of management components and their connections may exclude some operational configurations due to lack of knowledge in the management system about the status of the application components. However, it can never add more configurations, since they are a function only of the FTLQN model, not of the MAMA elements.

However, the addition of management components increases the total number of components (contributed by both MAMA and FTLQN) thereby increasing the number of states that have to be enumerated. Overall then, the state aggregation approach should have less computation than the enumeration approach.

4.1.1 Example

Let us consider the example of the layered console application shown in Figure 3.3 (Chapter 3) and a centralized management architecture managing the layered application as shown in Figure 3.9 (Chapter 3), in order to explain our solution approach. The probabili-

ties of being in failed state for each of the components (i.e. all the management tasks, application tasks and the processors) are assumed to be 0.1. Let us consider the mean total demand for execution on the processor for entries “CreateReport”, “Fetch”, “Read-A”, “GetApplicationData”, “GetConsoleData” and “LogReport” be 1 second and for the thin client entry “UI” to be 0.01 seconds. The execution demand for the backup entry “Read-B” is assumed to be 1.5 seconds. Let us further assume that on average, at every invocation, a caller entry makes one service request to each of the called entries.

The MAMA model in Figure 3.9 is first translated to the knowledge propagation graph shown in Figure 3.10 (Chapter 3). Next, combining the information of the FTLQN model of Figure 3.3 and the knowledge propagation graph of Figure 3.10, a non-coherent fault propagation graph (illustrated in Figure 3.11 of Chapter 3) is obtained. Then, the set of all distinct operational configurations and their associated probabilities are found, with the results shown in Table 4.1. For this system, the number of distinct operational configurations is much less than the number of system states, two vs. 2^{16} (=65536).

Table 4.1 Standby-Redundancy policy: Distinct Operational Configurations and their probabilities obtained using State Aggregation Approach

Configuration C_i	Number of system states mapped to C_i (as found from Enumeration approach)	Prob(C_i)	Reward R_i = mean throughput of task “Console”
C_1	16	0.2824	0.1996
C_2	10	0.0511	0.1815
System failed state	$65536 - (16 + 10) = 65510$	0.6665	0
where, C_1 implies “dbService is using server Database-A.” C_2 implies “dbService service is using server Database-B and Database-A failed.”			

An LQN model is created for each of the operational configurations, C_1 and C_2 , and solved using LQNS tool [FMNPRW96] to give their performance measures. In this exam-

ple, the mean throughput of the load-generating task “Console” has been selected to be the reward rate. Finally, the expected reward rate of the system is obtained as 0.0656 responses/sec.

If we change the redundancy policy of “dbService” from stand-by to load-balanced, we would get more operational configurations as shown in Table 4.2. This is because when both the servers, “Database-A” and “Database-B” are operational, “dbService” would use both of them with the workload being equally distributed among the two. The probabilities of the configurations are different in this case because of the following reasons: In the standby case, we assume that a standby server can be used if the failure status of the primary and the working status of the standby is known by the management system. However, in the load-balanced case, we assume that a redundant server can be used if its working status is known. The expected reward rate of the system in this load-balanced case is found to have increased to 0.0683 responses/sec, because the change in the knowledge function makes the system failed state less probable.

Table 4.2 Load-balanced policy: Distinct Operational Configurations and their probabilities obtained using State Aggregation Approach

Configuration C_i	Prob(C_i)	Reward R_i = mean <i>throughput</i> of task “Console”
C_1	0.0765	0.1996
C_2	0.0765	0.1815
C_3	0.2059	0.1901
System failed state	0.6411	0
where, C_1 implies “dbService is using server Database-A, Database-B failed” C_2 : implies “dbService is using server Database-B, Database-A failed” C_3 : implies “dbService service is using both Database-B and Database-A with equally distributed load.”		

4.2 Scalability Analysis (Comparison of Two Approaches)

This section demonstrates the efficiency of the *State Aggregation Approach* over the *Enumeration Approach* by comparing them through incremental scaling up of a layered application and its associated management system.

The example is a three-tier client-server system shown in Figure 4.2(a), with four tasks t1-t4 running on their own private processors (not explicitly shown in the figure). Task t4 is a backup of task t3. The associated management system is a centralized model with one central manager controlling the whole application. Figure 4.2(b) shows the centralized management model with no agents (the manager is directly watching all the application components). Figure 4.2(c) shows the same management model with agents running on each processor thereby increasing the number of management components (keeping the number of application components same). This is done in order to demonstrate the effect of increasing the number of management components on the performance of the two approaches.

Next, this application is scaled up in two stages; in the first stage (Figure 4.3) standby-servers are added in both layers and in the second stage (Figure 4.4) each service has two stand-bys with overlapping use. The management model (not shown here), with or without agents, for these two stages are also centralized and is similar to those used for the application in Figure 4.2(a), except that in these cases, the tasks interested in receiving notifications from the manager are t1, t2 and t5 for the application in Figure 4.3, and t1, t2, t5 and t8 for the application in Figure 4.4. The probabilities of being in failed state for all the components (i.e. all the management tasks, application tasks and the processors) are assumed to be 0.05. Let the mean total demand for execution on the processor for all the entries be 0.3 seconds except the thin client entry “e1” whose demand is assumed to be 0.01 seconds.

Figure 4.2 A three-tiered client-server system and its associated central management system (with agents in (b) and without agents in (c))

Figure 4.3 One stand-by server throughout

Figure 4.4 Two stand-by servers throughout

Scaled-up versions of the application in Figure 4.2. The corresponding MAMA models (with and without agents) would be similar to those of Figure 4.2 and are not shown here.

Table 4.3 shows the computation time taken for determining the distinct operational configurations and their probabilities by both the approaches. The computation time is in seconds and is measured on Windows98 hosted by Pentium (III) processor. Immediately we can see that aggregation is faster except for the smallest system, and that its effort grows more slowly as the system is made more complex.

For the management model with no agents, as we move from one stage to the next (see row-1, row-3 and row-5 in Table 4.3), we find an increase in the number of operational configurations due to the increasing addition of the stand-by servers, resulting in the generation of more non-coherent fault trees (one for each operational configuration), thereby increasing the computation time for the State Aggregation Approach. However, for a particular scaled stage of the application (e.g. see row-1 and row-2 in Table 4.3), as we add agents to the management model, the number of management components increases (consequently the number of nodes in the knowledge propagation graph gets larger) although the number of operational configurations remains the same. The increase in the CPU time

Table 4.3 Computation time for determining the distinct operational configurations by the Enumeration Approach and the State Aggregation Approach for the models in Figures 4.2 to 4.4

Application model	Centralized Management MAMA model	Number of System states (enumerated by Enumeration Approach)	Number of Operational Configurations (not including System failed state)	CPU time (in secs) for Enumeration Approach	CPU time (in secs) for State Aggregation Approach	Mean Throughput Reward (responses/sec)
Figure 4.2(a)	without agents (Figure 4.2(b))	1024	2	0.22	1.42	1.194
	with agents (Figure 4.2(c))	16384	2	1.49	1.48	1.075
Figure 4.3	without agents	16384	4	3.19	1.81	1.319
	with agents	1048576	4	102.65	2.36	1.125
Figure 4.4	without agents	1048576	9	252	2.97	1.334
	with agents	$\sim 5 \times 10^8$	9	> 12 hrs	5.16	1.138

for the State Aggregation approach in this case is due to the minpath determination algorithm which now works on a bigger knowledge propagation graph.

The Enumeration approach, however, performs badly compared to the State Aggregation Approach with the scaling up of application as well as management components. As we add more standby servers to the application, the number of reconfiguration points increases, resulting in a larger number of executions of the minpath algorithm. Therefore, although there is the same number of system states (that need to be enumerated by the Enumeration Approach) in row-2 and row-3 in Table 4.3, the computation time for the Enumeration approach is higher for the case in row-3.

4.3 A Comparison of Four Fault Management Architectures

This section studies the effect of four different fault management architectures on the expected reward rate of a distributed system. The four architectures were selected according to the classification given in [SRM95], for management systems based on the manager-agent paradigm.

Let us consider the FTLQN model shown in Figure 4.5. In this figure, there are 50 UserA users who could be working on one department of an enterprise, and 100 UserB users in another department. Each group makes primary access to applications through its departmental server (task AppA or AppB), which in turn accesses enterprise data servers Server1 and Server2. Server1 is the primary server to both departments; if it fails they will use Server2 until Server1 is working again.

Figure 4.5 A layered model of a client-server system with two groups of users. Server2 is the backup of Server1

For this example, the reward rate is selected as a weighted function of the average throughputs, as described in subsection 4.3.3.

4.3.1 Model Parameters

The failure and execution parameters for the model in Figure 4.5 are assumed as follows:

- UserA, UserB, procA and procB are perfectly reliable;
- the availabilities for all the other tasks and the processors are 0.9;

- the entries eA, eB, eA-1, eB-1, eA-2 and eB-2 have an execution demand (for their CPU time) with an average value of 1.0, 0.5, 1.0, 0.5, 1.0, 0.5 sec. respectively, each time they are invoked;
- each time an entry is invoked it makes 1 request (on average) to each entry it calls.

The following subsection describes the management architectures to be used with this system. Since the tasks UserA, UserB and their associated processors are assumed perfectly reliable, they are not monitored and are not shown in any of the management architectures described next.

4.3.2 Four sample fault management architectures

Architecture 1: A Centralized Management Architecture

The *centralized* architecture [CPW89], [Mars96] is the most commonly used. A single manager handles all agents and application tasks, makes the decisions and initiates reconfiguration. Figure 4.6 shows a centralized management architecture for the application in Figure 4.5. The central manager m1 manages all the tasks AppA, AppB, Server1, Server2 and their associated processors.

Let us consider the availability of the manager and all the agents be 0.9. In order to analyze the system in Figure 4.5 with the centralized management architecture in Figure 4.6, we first construct the knowledge propagation graph K from the MAMA model in Figure 4.6 and then we do the following:

For service, *serviceA*, in Figure 4.5, we compute the knowledge about the status of components Server1, Server2, proc3 and proc4 by the task AppA that requires serviceA from K as:

$$know_{Server1, AppA} = \bigwedge_{c = \{c3, ag3, c8, m1, proc5, c13, ag1, c5, AppA, proc1, proc3\}} o_c$$

(since there would be only one minpath from Server1 to AppA in K).

Figure 4.6 MAMA Model of a centralized management architecture for the layered application in Figure 4.5

$$know_{Server2, AppA} = \bigwedge_{c = \{c4, ag4, proc4, c10, m1, proc5, c13, ag1, c5, AppA, proc1\}} o_c$$

$$know_{proc3, AppA} = \bigwedge_{c = \{c7, m1, proc5, c13, ag1, c5, AppA, proc1\}} o_c$$

$$know_{proc4, AppA} = \bigwedge_{c = \{c9, m1, proc5, c13, ag1, c5, AppA, proc1\}} o_c$$

We repeat the steps for *serviceB*. Then, using these *know* functions and the information in Figure 4.5, the fault propagation graph *G* is generated. There are six operational configurations found:

C_1 implies Entry “userA” is operational and is using entry “eA” which in turn is using entry “eA-1” of task Server1. Entry “userB” is failed.

C_2 implies Entry “userA” is operational and is using entry “eA” which in turn is using entry “eA-2” of task Server2. Entry “userB” is failed.

C_3 implies Entry “userA” is failed. Entry “userB” is operational and is using entry “eB” which in turn is using entry “eB-1” of task Server1.

C_4 implies Entry “userA” is failed. Entry “userB” is operational and is using entry “eB” which in turn is using entry “eB-2” of task Server2.

C_5 implies Entry “userA” is operational and is using entry “eA” which in turn is using entry “eA-1” of task Server1. Entry “userB” is operational and is using entry “eB” which in turn is using entry “eB-1” of task Server1.

C_6 implies Entry “userA” is operational and is using entry “eA” which in turn is using entry “eA-2” of task Server2. Entry “userB” is operational and is using entry “eB” which in turn is using entry “eB-2” of task Server2.

An LQN model for each of the operational configurations is generated and solved using the LQNS tool for determining the performance measures. Table 4.4 shows the probability and the associated reward for each operational configurations.

Table 4.4 Configuration Probabilities (for Centralized Management) and Rewards

Configuration C_i	Perfect Knowledge Prob(C_i)	Centralized Management Prob(C_i)	Reward R_i = Total Throughput (A and B users)
C_1	0.125	0.117	0.5
C_2	0.024	0.021	0.5
C_3	0.125	0.117	1.0
C_4	0.024	0.021	1.0
C_5	0.531	0.314	1.11
C_6	0.100	0.057	1.11
System Failed	0.071	0.353	0

The expected steady-state reward rate for the system is obtained as approximately 0.619/secs whereas in case of perfect knowledge, it is found to be 0.924/secs.

To illustrate a situation with partial coverage of a failure, consider the failure of processor *proc3* (which supports *Server1*) when the system is fully operational. With perfect knowledge, this always leads to configuration C_6 . However under any management architecture, if agent *ag2* (which is connected to *AppB*) has also failed, then *AppB* does not know about the failure of *proc3* and does not reconfigure to use *Server2*; this leads to configuration C_2 instead. In configuration C_2 , the A group of users are operational and the B group are not. Thus the failure is partly covered and the system has reduced functionality.

This example also shows that the failures in the management architecture increase the probability of the system being failed or of reduced functionality.

Architecture 2: A Distributed Management Architecture

The *distributed* architecture [LF95] has multiple management domains with a manager for each one. When information from another domain is needed, the manager communicates with its peer systems to retrieve it. Figure 4.7 shows a distributed management architecture for the application in Figure 4.5. In Figure 4.7, there are two domain managers, *dm1* (for the entities *AppA*, *Server1*, *proc1* and *proc3*) and *dm2* (for the entities *AppB*, *Server2*, *proc2* and *proc4*). They communicate through the *notify* connections. Results are given in subsection 4.3.3.

Architecture 3: A Hierarchical Management Architecture

The *hierarchical* architecture [Herm90], [DSWY91] also relies on multiple managers (domain managers) and introduces the concept of the manager of managers (MOM). Each domain manager is connected to the rest of the network only through the MOM. The MOM operates on a higher hierarchical level, retrieving information from and coordinating the domain managers. Unlike the distributed architecture, the domain managers do not communicate directly. Figure 4.8 shows a hierarchical management architecture for the application in Figure 4.5, with two domain managers, *dm1* (for *AppA*, *Server*, *proc1* and *proc3*) and *dm2* (for *AppB*, *Server2*, *proc2* and *proc4*), and a manager of managers *mom1*. Results are given in subsection 4.3.3.

Figure 4.7 MAMA Model of a distributed management architecture for the application in Figure 4.5. dm1 and dm2 are two peer domain managers

Architecture 4: A General Network Management Architecture

The *network* architecture [Herm90] is a combination of the distributed and the hierarchical architectures. Instead of a purely peer or hierarchical structure, the managers are organized in a network. Figure 4.9 shows a network management architecture for the application in Figure 4.5. In Figure 4.9, there are two domain managers, dm1 and dm2, and two integrated managers, im1 and im2. dm1 manages the task Server1. dm2 manages the task Server2. im1 handles the task AppA, the managers dm1, dm2 and the processors proc3 and proc4. im2 handles the task AppB, the managers dm1, dm2 and the processors proc3 and proc4. The connections between managers can have any topology.

Figure 4.8 MAMA Model of a hierarchical management architecture for the application in Figure 4.5. mom1 is manager of managers handling both dm1 and dm2

4.3.3 Results

Suppose the agents and the managers in the four fault-management architectures described above have independent failures and failure probability 0.1. Define the reward rate R_i for configuration C_i as

$$R_i = w_{UserA} f_{i, UserA} + w_{UserB} f_{i, UserB}$$

where w_j represents the weight of users of group j and $f_{i,j}$ represents the throughput of users of group j corresponding to the configuration C_i . Table 4.5 shows the distinct operational configurations of the system, together with their probabilities and throughputs for the four fault management architectures and also for perfect knowledge.

Figure 4.9 MAMA Model of a network management architecture for the application in Figure 4.5.
im1, im2 are integrated managers

Figure 4.10 compares the expected steady state reward rate of the layered system in Figure 4.5 corresponding to each of the four management architectures, under variations of the weight of the UserB users relative to UserA. In Table 4.5, we observe that the variation in UserA throughput (given in second row from the bottom) for different management architectures is less than the variation in UserB throughput (given in the last row). As we increase the weight of UserB over UserA, the effect of UserB throughput on the reward rate of the system increases. UserB throughput decreases for the cases in Table 4.5 in the order Case 1, Case 5, Case 2, Case 3 and Case 4. Thus from Figure 4.10, we observe that the expected steady state reward rate of the system decreases for the architectures in the order network, centralized, distributed and hierarchical with the increase in the weight of UserB over UserA.

Figure 4.10 Comparison of expected steady state reward rate of the system in Figure 4.5 for the four management architectures

Table 4.5 Distinct Operational Configurations of the system in Figure 4.5, their probabilities for four fault management architectures and the associated throughput of two groups of users

i	Configuration C_i	Prob(C_i)					Throughputs of users ($f_{i, UserA}$, $f_{i, UserB}$)
		Perfect knowledge assumed (Case 1)	Central Architecture (Case 2)	Distributed Architecture (Case 3)	Hierarchical Architecture (Case 4)	Network Architecture (Case 5)	
1	C_1	0.125	0.117	0.176	0.225	0.148	(0.5, 0)
2	C_2	0.024	0.021	0.017	0.014	0.026	(0.5, 0)
3	C_3	0.125	0.117	0.095	0.076	0.148	(0, 1.0)
4	C_4	0.024	0.021	0.017	0.014	0.026	(0, 1.0)
5	C_5	0.531	0.314	0.254	0.206	0.282	(0.44, 0.67)
6	C_6	0.100	0.057	0.046	0.037	0.049	(0.44, 0.67)
	System Failed Configuration	0.071	0.353	0.395	0.428	0.321	(0, 0)
	Average UserA throughput	0.352	0.232	0.228	0.226	0.233	
	Average UserB throughput	0.572	0.387	0.313	0.253	0.396	

The algorithm costs are different for analyzing the different architectures. For the five cases given in the table, the number of states in the solution state space is 256, 16384, 65536, 262144 and 65536 respectively. The larger state spaces arise for systems with more management components.

4.4 Best Placement of Managers

This section considers how managers should be placed in the system of Figure 4.5. Two managers are placed in eight different ways, and the throughputs of the two classes of users are determined, for the parameters used in subsection 4.3.1. There are large differences between the results for different placements, which may favor one class of users or the other. The best placement depends on the relative rewards for the two classes.

Figures 4.12-4.19, which show the eight placements of the two managers, use a shorthand for the MAMA notation to simplify the presentation. Each processor has a particular application task and an agent, as shown in Figure 4.11. The application task and its interactions with the agent are omitted in the shorthand, while the agent is shown as a grayed rectangle so that connectors can be attached to it. In the shorthand each processor is shown with its agent and possibly a manager task, and the connectors between them.

The eight cases of alternative placements are described next, with the results collected together in Table 4.6.

Case 1 (Figure 4.12): The two managers, m1 and m2, are placed in the caller processors, proc1 and proc2. Both the agents, ag3 and ag4, are *status-watched* by both the managers in order to detect their failure and also to know about the status of the servers, Server1 and Server2 respectively. Both proc3 and proc4 are also watched for liveness by the two managers in order to distinguish between the processor failure or the corresponding agent failure. m1 also *status-watches* the agent ag1 to know about the status of the application task AppA. The domain monitored by m1, called Domain-1, includes the components proc3, proc4, Server1, Server2 and AppA. Domain-1 is shown surrounded by a dashed closed curve in this figure. Similarly, the domain monitored by manager m2, called Domain-2,

includes the components proc3, proc4, Server1, Server2 and AppB and is shown surrounded by a solid closed curve. In this architecture, the caller processors, proc1 and proc2 (which do not have any backups) are not watched for liveness detection.

Case 2 (Figure 4.13): Manager task m2 is moved from the caller processor proc2 (as in Figure 4.12), to the primary server's processor, proc3. The domain of m2 now includes both the primary and the backup servers. Manager m1 now obtains the status information about the backup server from manager m2 and watches the primary server processor proc3 as well. In this architecture, if m2 fails, the applications AppA or AppB would not have any knowledge about the status of either the primary or the backup server unlike in Case 1, where m2's failure would not affect the knowledge of AppA about the servers.

Case 3 (Figure 4.14): This is a variation on Case 2, in which the manager m2 is moved again, from proc3 to proc4. This case evaluates the effect of moving m2 to the backup server's processor.

Case 4 (Figure 4.15): This is another variation on Case 2. Here, the manager m1 has been moved to proc2 (AppB's processor) from proc1 (AppA's processor) while m2 stays in proc3. Since the execution demand of AppA from the servers is larger than that of AppB (refer subsection 4.3.1 for the execution demands), moving the manager away from AppA will reduce the workload on proc1, compared to Case 2. However it also changes the failure detection probabilities.

Case 5 (Figure 4.16): This is a variation of Case 3 in Figure 4.14. Here the manager m1 has been moved from proc1 to proc2 and m2 stays back in proc4. As in Case 4, this move reduces the pressure on processing in proc1, and also affects the failure detection compared to Case 3.

Case 6 (Figure 4.17): The managers tasks now have their own dedicated processors. The management domains are separate and are divided horizontally; m1 handles a domain containing the callers, and m2 handles a domain containing the servers. m1 and m2 communicate with each other through *notify* connections.

Case 7 (Figure 4.18): This is a variation on Case 6 where the management domains are divided vertically. Manager m1 now handles the domain containing AppA, the primary server and their processors while m2 handles the domain containing AppB, the backup server and their processors.

Case 8 (Figure 4.19): Both the managers watch all the application tasks and their processors, as if manager m2 is a replica of m1.

Figure 4.11 Shorthand notations for the deployment of agents, and for the connectors, as used in Figures 4.12-4.19

Figure 4.12 Case-1: Placement of the managers m1 and m2 in callers' processors proc1 and proc2. m1 manages Domain-1 and m2 manages Domain-2

Figure 4.13 Case-2: Placement of the managers m1 in proc1 and m2 in proc3 (i.e. the primary server's processor)

Figure 4.14 Case-3: Placement of the managers m1 in proc1 and m2 in proc4 (i.e. the backup server's processor)

Figure 4.15 Case-4: Placement of the managers m1 in proc2 and m2 in proc3 (variation of Figure 4.13)

Figure 4.16 Case-5: Placement of the managers m1 in proc2 and m2 in proc4 (variation of Figure 4.14)

Figure 4.17 Case-6: Placement of the managers m1 and m2 on their dedicated processors. Domains are setup horizontally

Figure 4.18 Case-7: Placement of the managers m1 and m2 on their dedicated processors. Domains are setup vertically

Figure 4.19 Case 8: Placement of the managers m1 and m2 on their dedicated processors. m1 is managing the same set of components as being managed by m2

Table 4.6 shows the probabilities of different operational configurations of the system in Figure 4.5 and the associated throughput of two groups of users (UserA and UserB) for different manager placements. The availabilities of all the components have been assumed to be 0.9 except UserA, UserB, procA and procB which are assumed to be fully reliable.

From Table 4.6, it is seen that even though the configuration probabilities are different, the user throughputs are the same in cases 2 and 3 and also in cases 4 and 5. This shows that it does not matter whether m2 is placed either on the primary or backup server's processor (proc3 or proc4). In Cases 6 and 7, the throughput of UserB is the same but for UserA it is higher in Case 7. This is due to the fact that, in Case 6, the probability of configuration C_1 (UserA using primary Server1, UserB failed) is lower than in Case 7 since the knowledge information about Server1 to application AppA (which is used by

Table 4.6 Probabilities of Operational Configurations of the system in Figure 4.5 for different manager placements and the associated throughput of two groups of users (UserA and UserB)

Case	No. of states enumerated	Probabilities of Operational Configurations C_i (Total Number of reachable configurations of tasks = 6)							Average UserA throughput*	Average UserB throughput**
		C_1	C_2	C_3	C_4	C_5	C_6	$System_{failed}$		
1	16384	0.164	0.030	0.164	0.030	0.314	0.057	0.241	0.260	0.443
2		0.117	0.009	0.074	0.006	0.314	0.025	0.455	0.212	0.306
3		0.105	0.021	0.066	0.013	0.282	0.057	0.455	0.212	0.306
4		0.074	0.006	0.117	0.009	0.314	0.025	0.455	0.189	0.353
5		0.066	0.013	0.105	0.022	0.282	0.057	0.455	0.189	0.353
6	65536	0.094	0.017	0.094	0.017	0.254	0.046	0.478	0.188	0.313
7		0.176	0.017	0.094	0.017	0.254	0.046	0.395	0.229	0.313
8		0.139	0.025	0.139	0.025	0.373	0.068	0.231	0.276	0.459
Throughputs of users $(f_{i,UserA}, f_{i,UserB})$		(0.5, 0)	(0.5, 0)	(0, 1)	(0, 1)	(0.44, 0.67)	(0.44, 0.67)	(0, 0)		

where,

C_1 implies Entry "userA" is operational and is using entry "eA" which in turn is using entry "eA-1" of task Server1. Entry "userB" is failed. (Includes two reachable configurations of tasks corresponding to Server2 being up or failed)

C_2 implies Entry "userA" is operational and is using entry "eA" which in turn is using entry "eA-2" of task Server2. Entry "userB" is failed.

C_3 implies Entry "userA" is failed. Entry "userB" is operational and is using entry "eB" which in turn is using entry "eB-1" of task Server1. (Includes two reachable configurations of tasks corresponding to Server2 being up or failed)

C_4 implies Entry "userA" is failed. Entry "userB" is operational and is using entry "eB" which in turn is using entry "eB-2" of task Server2.

C_5 implies Entry "userA" is operational and is using entry "eA" which in turn is using entry "eA-1" of task Server1. Entry "userB" is operational and is using entry "eB" which in turn is using entry "eB-1" of task Server1. (Includes two reachable configurations of tasks corresponding to Server2 being up or failed)

C_6 implies Entry "userA" is operational and is using entry "eA" which in turn is using entry "eA-2" of task Server2. Entry "userB" is operational and is using entry "eB" which in turn is using entry "eB-2" of task Server2.

$$\text{*Average UserA throughput} = \sum_{i=1}^6 f_{i, UserA} \cdot Prob(C_i)$$

$$\text{** Average UserB throughput} = \sum_{i=1}^6 f_{i, UserB} \cdot Prob(C_i)$$

UserA) has to propagate through both m2 and m1 in Case 6 whereas in Case 7, the knowledge has to propagate only through m1 (which in turn increases the probability that AppA will have knowledge about Server1).

Case 8 gives the highest throughputs for both groups of Users, with Case 1 a close second. If the cost of the two dedicated processors needs in Case 8 is too high to be justified by the benefit, then Case 1 (managers placed in caller processors) appears to be the best.

Similar to subsection 4.3.3, let us define the reward rate R_i for configuration C_i as:

$$R_i = w_{UserA} f_{i, UserA} + w_{UserB} f_{i, UserB}$$

where w_j represents the weight of users of group j and $f_{i,j}$ represents the throughput of users of group j corresponding to the configuration C_i . If we assume that this weight factor is associated according to the priority of the corresponding user group, we can then develop (using Table 4.6) two types of rankings among the different placements, depending on the throughput of the higher priority user, as follows:

- (i) When $w_{UserA} \gg w_{UserB}$, i.e. UserA has much higher priority than UserB, then the ranking would be in order Case 8, Case 1, Case 7, Case 2 or 3, Case 4 or 5, Case 6.
- (ii) When $w_{UserB} \gg w_{UserA}$, i.e. UserB has much higher priority than UserA, then the ranking would be in order Case 8, Case 1, Case 4 or 5, Case 6 or 7, Case 2 or 3.

This approach to management modelling is only limited by the assumption that the failures of the management components might inhibit a caller from using a service even when the service is working because the caller lacks the knowledge of the service's status.

4.5 Summary

This chapter has introduced two solution strategies, called *Enumeration* and *State-Aggregation*, to compute the steady-state and the point performability from the Dependable-LQN model. The enumeration approach, which is a bottom-up approach, involved enumerating all the system configuration states (i.e. 2^N states where N is the total number of components including the management components) that is equal to the number of states

in the underlying Markov Reward model. The State Aggregation approach, which is a top-down approach, generated the operational configurations and their probabilities without generating or enumerating all the states. Both the strategies utilize minpath generation algorithms where the number of minpaths between two nodes may be exponential in the number of nodes in the Knowledge Propagation graph. In SAA approach, the number of operational configurations increases with the number of services depending on the redundancy policy of the services. If the policy is primary-backup, the increase is linear, whereas if the policy is load-balanced or active or primary-standby-active, the increase is exponential. In the worst case, the number of operational configurations can grow up to maximum $2^{M/2}$ where M is the total number of components in the FTLQN model that does not include the components in the MAMA model. Both the solution algorithms have been published in [DW04a].

This chapter demonstrated how the Dependable-LQN model can be used to compare different alternatives of management architectures. From the results obtained in Section 4.3, it was found that a general network management architecture performed better than the centralized, hierarchical or a distributed architecture for the system in Figure 4.5. These results have been published in [DW02].

This chapter also showed how the model can be used to compare different placements of managers. It has been found (in Section 4.4) that the best placements of two managers for the system in Figure 4.5 was when the managers are replicas and both of them are watching the same domain. These results have been published in [DW04b].

Chapter 5

The Dependable-LQN Model Solver

This chapter presents a high level description of the *DependableLQNS* tool that automates the solution of a Dependable-LQN model. This tool implements both the *State-Aggregation Approach* and the *Enumeration Approach*. This tool has a command-line interface, called *dlqns*, and a graphical user interface, called *dlqnsGUI*.

Section 5.1 presents the high level description of the tool. Section 5.2 describes how to run the tool from the command-line and Section 5.3 describes how to use the GUI.

5.1 Description of the DependableLQNS tool

A Dependable-LQN model can be described in either a plain text file or it can be specified by using the GUI that automatically generates the corresponding text file. To describe the model in a text file, an architecture description language, called *MAMA-dl*, has been developed as part of this work. The BNF description of this language has been provided in Appendix A. The parser for this language allows a user to:

- describe the application and management components (processors, application software tasks, manager tasks, agent tasks) and pseudo-components representing additional shared factors of failures (such as power-supply failure, failure due to mechanical vibrations).
- describe the connectors interconnecting the components in the management architecture.
- specify the failure and service dependencies among the application components from layer to layer.
- specify performance and dependability parameters (for example, execution demand of an entry on its processor, number of service requests made by an entry to the other

from layer to layer, failure and repair rates for the components and the redundancy mechanisms employed).

Figure 5.1 shows the sequence of processing steps that are carried out by the tool. The MAMA-*dl* parser reads the text file containing the model description and it creates the MAMA and FTLQN model elements. The tool then generates the fault propagation graph and knowledge propagation graph from the FTLQN and the MAMA models and determines the distinct operational configurations of the application from the information embedded in those graphs. If Enumeration Approach is selected as the solution algorithm, then the tool computes the probabilities of the operational configurations by enumerating all the system states. Otherwise, if the State Aggregation Approach is selected, the tool generates non-coherent fault trees, one for each operational configuration, and solves them using the Aralia tool [DR97] to compute the probabilities. Next, the tool generates an LQN model for each operational configuration and solves them using the LQNS tool [FMNPRW96]. Finally it computes the expected reward rate by combining the configuration probabilities and configuration rewards. The graphical user interface of the tool has been written in Java while the rest of it has been written in C++ for Windows.

The output of the tool consists of the different operational configurations, their probabilities, their associated reward rates and the average reward for the whole application. Presently, the mean throughput of the pure client task is taken as the reward rate for a particular configuration.

5.2 Running the tool from the command-line

The command-line interface of the tool can be invoked by typing the following command in a DOS-window and then pressing Return:

```
% dlqns.exe a|e pathOfAraliaExecutable pathOfLqnsExecutable filename
```

The first argument can be either “a” or “e” where the option “a” is for invoking *State-Aggregation Approach* and the option “e” is for invoking *Enumeration Approach*. The second and third arguments specify where the Aralia and the LQNS executables reside in

the system. The fourth argument, *filename*, should have .dlqn extension, for example, *foo.dlqn*. The model description in the file should follow the MAMA-dl language described in Appendix A.

A sample usage is demonstrated using the example shown in Figure 5.2. The input file *simple-example.dlqn* corresponding to the models in Figure 5.2 is shown in Figure 5.3. In Figure 5.3, Lines 6 through 10 describe the five processors, Lines 14 through 22 describe all the tasks, Lines 27 through 40 describe the connectors of the MAMA model, Lines 44 through 47 describe the CPU demand for the four entries and Lines 48 and 49 describe the service dependencies among the entries. In this figure, the connectors have failure rate and repair rate equal to zero. It implies they are fully reliable. This example has two operational configurations, one is where Entry2 is using the primary Entry3 and the other is where Entry2 is using the backup Entry4.

The results obtained by solving the model described in the file *simple-example.dlqn* are shown in Figure 5.4. In this figure, there are some elements in the operational configurations that are not specified by the user, for example *service2_0*. These are the dummy elements that are generated by the tool while creating the Fault Propagation graph.

5.3 Running the tool using the Graphical User Interface

The graphical-user interface of the tool can be launched either by double-clicking the icon whose target executable is *dlqnsGUI.bat* or by typing the following command in a DOS-window and then pressing Return:

```
% dlqnsGUI.bat
```

A window appears as shown in Figure 5.5. To open an existing file, choose File > Open as shown in Figure 5.6 and then locate the file that you want to open and then click Open. To create a new file, choose File > New. To save an opened file, click File > Save and to close an opened file, click File > Close.

Figure 5.2 Dependable-LQN model of a simple example

```

1  G "A simple Dependable-LQN model" 1.0E-6 50 5 0.9 ;
2  component_description
3  {
4      proc_description
5      {
6          p Processor1 0.05 0.95 f ;
7          p Processor2 0.05 0.95 f ;
8          p Processor3 0.05 0.95 f ;
9          p Processor4 0.05 0.95 f ;
10         p Processor5 0.05 0.95 f ;
11     }
12     task_description
13     {
14         t Task1 AT Processor1 0.05 0.95 r [ Entry1 ] m 1 ;
15         t Task2 AT Processor2 0.05 0.95 n [ Entry2 ] m 1 ;
16         t Task3 AT Processor3 0.05 0.95 n [ Entry3 ] m 1 ;
17         t Task4 AT Processor4 0.05 0.95 n [ Entry4 ] m 1 ;
18         t Agent1 AGT Processor1 0.05 0.95 ;
19         t Agent2 AGT Processor2 0.05 0.95 ;
20         t Agent3 AGT Processor3 0.05 0.95 ;
21         t Agent4 AGT Processor4 0.05 0.95 ;
22         t Manager1 MT Processor5 0.05 0.95 ;
23     }
24 }
25 mama_model
26 {
27     c c1 AW ( t , Task1 ) ( t , Agent1 ) 0.0 0.0 ;
28     c c2 AW ( p , Processor1 ) ( t , Manager1 ) 0.0 0.0 ;
29     c c3 SW ( t , Agent1 ) ( t , Manager1 ) 0.0 0.0 ;
30     c c4 AW ( t , Task2 ) ( t , Agent2 ) 0.0 0.0 ;
31     c c5 AW ( p , Processor2 ) ( t , Manager1 ) 0.0 0.0 ;
32     c c6 SW ( t , Agent2 ) ( t , Manager1 ) 0.0 0.0 ;
33     c c7 AW ( t , Task3 ) ( t , Agent3 ) 0.0 0.0 ;
34     c c8 AW ( p , Processor3 ) ( t , Manager1 ) 0.0 0.0 ;
35     c c9 SW ( t , Agent3 ) ( t , Manager1 ) 0.0 0.0 ;
36     c c10 AW ( t , Task4 ) ( t , Agent4 ) 0.0 0.0 ;
37     c c11 AW ( p , Processor4 ) ( t , Manager1 ) 0.0 0.0 ;
38     c c12 SW ( t , Agent4 ) ( t , Manager1 ) 0.0 0.0 ;
39     c c13 NTFY ( t , Manager1 ) ( t , Agent2 ) 0.0 0.0 ;
40     c c14 NTFY ( t , Agent2 ) ( t , Task2 ) 0.0 0.0 ;
41 }
42 ftlqn_model
43 {
44     s Entry1 0.01 0.0 0.0 ;
45     s Entry2 0.3 0.0 0.0 ;
46     s Entry3 0.3 0.0 0.0 ;
47     s Entry4 0.3 0.0 0.0 ;
48     y Entry1 Entry2 1.0 0.0 0.0 ;
49     ALT y Entry2 ( service2 [ Entry3 , Entry4 ] ) 1.0 0.0 0.0 ;
50 }

```

Figure 5.3 Input file *simple-example.dlqn* corresponding to the model in Figure 5.2

```

CONFIGURATIONS:
cfg2: [ (Entry1,1) (Entry2,1) (service2,1) (service2_0,1) (Entry3,1)
        (Task3,1) (Processor3,1) (service2U_Entry3,1) (Processor2,1)
        (Task2,1) (Task1,1) (Processor1,1) ]
Probability: 0.598737
Throughput of Entry1: 1.63934

cfg3: [ (Entry1,1) (Entry2,1) (service2,1) (service2_1,1) (Task2,1)
        (Processor2,1) (Entry4,1) (Task4,1) (Processor4,1) (Task1,1)
        (service2U_Entry4,1) (Processor1,1) (service2D_Entry3,1) ]
Probability: 0.0569549
Throughput of Entry1: 1.63934

Total number of configurations: 2
Expected Throughput of entry Entry1: 1.0749
Steady-state Availability of entry Entry1: 0.655692

Program successfully terminated.
Elapsed Time: 0.156 seconds.

```

Figure 5.4 Output generated by *dlqns* for the input in Figure 5.3

Figure 5.5 The GUI window

Figure 5.6 Opening an existing file

Figure 5.7 shows the GUI window when the file *simple-example.dlqn* is opened. There are six tabs for describing the six model elements: processors, application tasks, entries, services, management tasks and connectors. Each tab has columns corresponding to the attributes of the model element. For example, a processor has attributes: name, failure rate, repair rate and scheduling discipline. To add, edit or delete a model element, choose Add, Edit or Delete buttons at the bottom of the window in the corresponding tab.

Figure 5.7 The file *simple-example.dlqn* is opened

On selecting the row whose “Name” attribute is “Processor2” and then clicking the Edit button, a new window titled “Edit Processor” appears as shown in Figure 5.8. Figures 5.9-5.13 show the windows for editing the Connectors, Application tasks, Entries, Services and Management tasks. For *Connectors*, we need to provide its name, type (Alive-watch, Status-watch or Notify), source component, destination component, failure rate and its repair rate. For *Application Tasks*, we need to specify its name, failure rate, repair rate, number of threads, is it reference task or not and its processor. For *Entries*, we need to provide its name, its task and the service times (i.e. its CPU demand) for the three phases where the first phase of service is before sending the reply, and an optional second and third phases are after the reply. While in the first phase, the client task making the request is blocked and the server merely continues the thread of control of its client, but in

the second and third phases it has an independent thread of control of its own. For *Services*, we need to provide its redundancy policy, source entry, target entries and the mean number of calls made to the target(s) in three phases. For *Management Tasks*, we need to provide its name, is it an agent or a manager, failure rate, repair rate and its processor.

Edit Processor

Name: Processor2

Failure Rate: 0.05

Repair Rate: 0.95

Scheduling Discipline:

- ☒ First come first served
- ☐ Priority Preemption
- ☐ Head of Line
- ☐ Random

OK Cancel

Figure 5.8 Editing “Processor2”

Edit Connector

Name: c1

Type: Alive Watch

Source: Task1

Destination: Agent1

Failure Rate: 0.0

Repair Rate: 0.0

OK Cancel

Figure 5.9 Editing connector “c1”

Edit Application Task

Name: Task1

Failure Rate: 0.05

Repair Rate: 0.95

Number of Threads: 1

☐ Infinite Threads

☒ Is Reference Task

Processor: Processor1

OK Cancel

Figure 5.10 Editing “Task1”

Edit Entry

Name: Entry1

Phase-1 Service Time: 0.01

Phase-2 Service Time: 0.0

Phase-3 Service Time: 0.0

Task: Task1

OK Cancel

Figure 5.11 Editing “Entry1”

Figure 5.12 Editing “service2”

Figure 5.13 Editing the management task “Agent1”

To create a new file, choose File > New and then name the file with a .dlqn extension, for example, *foo.dlqn*. Fill in data about the model elements by selecting the “Add” button at the bottom of the window for each tab. Then save the file by choosing File > Save. On clicking the Add button in the “Processors” tab, a new window titled “Add Processor” appears as shown in Figure 5.14. The Add windows for other model elements are not shown here. They are very similar to their Edit windows.

Figure 5.14 Adding a processor

A model can be solved by choosing Solve > Start. A new window appears showing the

results as given in Figure 5.15. To edit the parameters needed by the LQNS tool, choose Solve >Edit Solver Parameters.

Figure 5.15 The Results window

The GUI of the tool does not support the specification of the pseudocomponents and the specification of the AND-OR *depends* relationships that may exist between entries and pseudocomponents to describe only failure dependencies. These features are supported by the command-line interface, *dlqns*.

5.4 Summary

This chapter has described the processing steps that are carried out by the DependableLQNS tool and the relationships between them. It described a GUI for the tool that makes it easier to use. A description of this tool has been published in [DW03b].

Chapter 6

Case Study: An Air Traffic Control System

An Air Traffic Control (ATC) system [CDD96], [BCK98], [DDMS95] is a large-scale complex distributed system which demands high availability and high performance. Its software architecture plays a key role in achieving its high quality requirements.

We consider the system in [CDD96]. This is an en route system which controls aircraft from soon after takeoff until shortly before landing. Its end users are the air traffic controllers. It has a layered software architecture with one subsystem depending on another for services and it uses a separate management architecture for automatic failure detection and recovery. It is an important case study for this research because the Dependable-LQN model perfectly fits the choice for modelling such a complicated and large system.

The goal of this chapter is to demonstrate the use of the Dependable-LQN model on a substantial system with strong requirements for performance and dependability. It also describes the use and scalability of the DependableLQNS tool described in Chapter 5.

6.1 Dependable-LQN Model of an ATC En Route System

The system has the following general description [CDD96].

An airspace controlled by a single ATC facility is administratively divided into sectors. For example, US airspace is serviced by 20 facilities, with up to 118 sectors per facility. Each facility receives aircraft surveillance and weather data from radars and communicates with other external subsystems such as other en route facilities. Inside each facility, air traffic services are divided among four subsystems:

- Surveillance Processing Service (that receives radar data and correlates it with individual aircraft tracks) is provided by the Radar subsystem directly connected to radars,
- Flight Plan processing and Conflict Alert services are provided by the Central sub-

system,

- Display service (which displays aircraft position information obtained from radars and allows inputs from air traffic controllers to modify flight plan data or change display format) is hosted by the Console subsystem,
- Monitoring service (which provides the monitoring and control service for other ATC services ensuring their availability policies) is hosted by the Monitor and Control subsystem.

There are up to four consoles allocated for handling each sector. Fault-tolerance is achieved by software server groups. For example, there are up to four Display Management load-balanced redundant servers per sector, three primary/standby active redundant Surveillance Processing servers, two primary/standby Flight Plan Management servers.

All the three redundant Surveillance Processing servers receive the raw radar data from the radars in parallel, however, only the primary sends the processed radar data to the Display Management servers running in the consoles. A PSAR redundancy policy is used to model this case.

Figure 6.1 shows some parts of a Dependable-LQN model for a single sector, based on the description in [CDD96]. Each dotted bubble represents a process group with replication; the service nodes and redundant servers are not shown. The remainder of the model is described in additional Figures 6.2-6.4. The communications with replicas is made transparent by a process-to-process session manager (P2PSM) in each host (which is in the model but not shown in Figure 6.1). Figure 6.2 shows the communication between the task DisplayManagement and the replicas of FlightPlanManagement through a P2PSM. Other communications with replicas follow the same pattern and are not shown here.

Some failure dependencies are implicit in the server request-reply dependencies. Others can be made explicit with a *depends* relationship, for instance that “display radar data” depends on “process radar data”, even though the communications is asynchronous. This is an OR depends relationship on the three radar processing replicas, since any one is sufficient. The OR relationship is indicated in the Figure 6.1.

Figure 6.1 A Dependable-LQN model for an ATC en route system. The redundant server groups are not shown here. The dotted boundary indicates one replica of a replicated subsystem

Figure 6.2 Portion of MAMA model for Figure 6.1. Communication between the tasks DisplayManagement and the replicas of FlightPlanManagement is made transparent by the P2PSM task residing in the consoleProc

The fault management architecture depends on a *group availability management* server (gSAM) on each processor which monitors all the software servers in its own processor and also monitors the other processors in its software server group. In MAMA terms, the gSAM servers maintain an *alive-watch*. This is supported by three redundant name servers which maintain a list of primary host processors for each server group. When a failure of a software server occurs in its processor, a gSAM server notifies other gSAM servers in its own group. Similarly, the failure of a processor is detected by the other gSAM servers in a group. Whenever a failure is detected, the gSAM server notifies the name servers which in turn notify the relevant P2PSM's so that they can retarget their service requests.

Figure 6.3 shows a portion of the MAMA model for Figure 6.1, including one replica of the Monitor and Control subsystem. The redundant servers are not shown here.

Figure 6.4 shows the interactions among the gSAM servers in the central subsystem group containing two replicas. The interactions among the gSAM servers in other groups are similar to this one and are not shown here.

Figure 6.3 Portion of MAMA model for Figure 6.1. Redundant server groups and interactions among the gSAM servers in a group are not shown here

Figure 6.4 Portion of MAMA model for Figure 6.1 showing interactions among the gSAM servers in a group of two replicas of the central subsystem

6.2 Numerical Results

This section analyzes the Dependable-LQN model of the ATC system which is described in Section 6.1. The model is parameterized in subsection 6.2.1 and the average throughput and average response time for the Controller task is computed in subsection 6.2.2. Subsection 6.2.3 shows the effect of failures of management components on system availability. Subsection 6.2.4 identifies the most-sensitive and least-sensitive component. Subsection 6.2.5 determines an optimal replication number for the manager (i.e. the name server) for a given availabilities for management components. Subsection 6.2.6 assumes unreliable

agents (i.e. the gSAM servers) and shows the effect of replication of agents on system availability.

6.2.1 Parameterizing the model

Parameter values for this system were invented, including the failure and repair rates of the components, CPU execution demand for the entries and the mean number of service requests made from one entry to another per call.

The mean time to failure (MTTF) and mean time to repair (MTTR) for all the processors are 1 year and 2 days respectively so that their steady-state availability is approximately 0.99455. The MTTF and MTTR for all the tasks except the gSAM servers are 30 days and 1 hour respectively so that their steady-state availability is approximately 0.99861. For the gSAM servers, the MTTF is 15 days and MTTR is 1 hour so that their availability is approximately 0.99723. The tasks “Controller”, “Radars” and their associated processors are assumed to be fully reliable.

The execution demand for the entries of tasks “UI” and “DisplayManagement” are assumed to be 0.01 seconds, of tasks “FlightPlanManagement” and “FlightPlanDatabase” are assumed to be 0.02 seconds, of task “SurveillanceProcessing” is assumed to be 0.5 seconds and of tasks “ConflictResolution” and “TrajectoryManagement” are assumed to be 0.001 seconds. The CPU demand for all the entries of the P2PSM tasks (not shown in Figure 6.1) are assumed to be 0.00001 seconds. In average, the entry “userInterface” makes 1 call to “modifyDisplay”, 1 call to “modifyFlightPlan” and 2 calls to “displayFlightPlan”, per invocation. When the rest of the entries are invoked they make 1 request (in average) to each entry they call.

6.2.2 Results on Computational Effort

The model for a sector (as illustrated in Figures 6.1-6.4) has 13 processors and 38 tasks (including the P2PSM and gSAM tasks), i.e. 51 components. There are 114 connectors in the MAMA model. On running the model by DependableLQNS tool using the *State-Aggregation Approach*, we obtain:

Number of Operational Configurations:	14
Number of nodes generated in Knowledge Propagation graph:	98
Number of nodes generated in Fault Propagation graph:	714
Average Throughput of Controller task:	1.5376 requests/sec
Average response time of Controller task:	1.951 seconds
Probability that all the three “userInterface” entries are working: (there are two operational configurations for this case)	0.947714
Probability that two “userInterface” entries are working: (there are six operational configurations for this case)	0.035460
Probability that only one “userInterface” entry is working: (there are six operational configurations for this case)	0.000246
System availability:	0.983416
Model solution time (on Windows XP hosted by Pentium IV, 2.66 GHz processor):	76.18 seconds

6.2.3 Effect of management component failures on System Availability

In order to see the effect of the management component failures on system availability, we varied the availability (from 0.99 to 1) for all the management components and observed the effect on system availability. The management components include the name servers and their associated processors, and the gSAM servers. In the base case, the availabilities for the name servers, the processors for the name servers, and the gSAM servers were 0.99861, 0.99455 and 0.99723 respectively.

Figure 6.5 shows the system availability as a function of the availability of management components. From this figure, it can be seen that the system availability increases linearly with the availability of management components. It increases up to 0.9869 where the management components are assumed fully reliable. If the system availability requirement is greater than 0.9869, then this result suggests that solely increasing the manage-

ment component availabilities would not achieve the goal, it would require changes in the application component availabilities and their replication strategies.

Figure 6.5 The predicted System Availability as a function of the availabilities of the management components

6.2.4 Sensitivity of System Availability to all components

In order to see which component is the strongest determinant of the system availability, we decreased the availability of each component in turn by 10 percent and observed the effect on the predicted system availability.

The sensitivity of the predictions to a ten-percent change in the availabilities is shown in Table 6.1. It can be seen that the system availability is most sensitive to the change in the availabilities of the processors and the gSAM servers of the Central subsystem. A ten-percent decrease at the gSAM servers in Radar processors has negligible effect on system availability. This is because the knowledge (propagated through the management architecture) of the Surveillance Processing servers running on the Radar processors is not critical to the retargettable services of the system.

Table 6.1 Sensitivity of System Availability to availabilities of components

Parameter	System Availability	% change in System Availability
No change, i.e. Original Model	0.983416	
10% decrease in availability of either {consoleProc1, consoleProc2, consoleProc3, Console1-gSAM, Console2-gSAM, Console3-gSAM}	0.981828	-0.1615 %
10% decrease in availability of either {centralProc1, centralProc2, Central1-gSAM, Central2-gSAM}	0.885075	-9.999938 %
10% decrease in availability of either {radarProc1, radarProc2, radarProc3, SP1, SP2, SP3, Radar1-P2PSM, Radar2-P2PSM, Radar3-P2PSM}	0.98341	-6.101×10^{-4} %
10% decrease in availability of either {nsProc1, nsProc2, nsProc3, NameServer1, NameServer2, NameServer3}	0.983412	-4.067×10^{-4} %
10% decrease in availability of either {UI1, DM1, UI2, DM2, UI3, DM3, Console1-P2PSM, Console2-P2PSM, Console3-P2PSM}	0.983408	-8.135×10^{-4} %
10% decrease in availability of either {FPM1, FPDB1}	0.982741	-0.068638 %
10% decrease in availability of either {CR1, TM1, FPM2, CR2, FPDB2, TM2, Central1-P2PSM, Central2-P2PSM}	0.982742	-0.068537 %
10% decrease in availability of either {Radar1-gSAM, Radar2-gSAM, Radar3-gSAM}	0.983416	no change

The sensitivity results suggest that it would be better to improve the availability of the processors and the gSAM servers of the Central subsystem than any other component to increase the system availability.

6.2.5 Optimal Replication Number for Name Servers

For a given system availability goal, we would like to answer questions such as: whether replication of name servers (and their associated processors) would help to achieve that goal and if yes, how many replicas would be needed.

We varied the availability and replication of the NameServer and its associated processor, nsProc, and observed its effects on the system availability, with results shown in Table 6.2. From row-1 of this table, we see that when the NameServer and its processor are fully available, the system availability is 0.983417. This suggests that if the system availability requirements are greater than 0.983417, then replicating the NameServer would not help. Let us suppose that the requirement is 0.9832. Then, from row 2 to 5 of the table, it can be seen that

- if the availabilities of NameServer and nsProc are 0.99, then three replicas are needed to meet the goal
- if the availabilities of NameServer and nsProc are 0.999 or if they are same as assumed in subsection 6.2.1, then two replicas are needed to meet the goal
- if the availabilities of NameServer and nsProc are 0.9999, then one replica would suffice

Table 6.2 System Availability for different number of NameServer replicas

Availability of <i>NameServer</i>	Availability of <i>nsProc</i>	System Availability for 1, 2, and 3 replicas of <i>NameServer</i> and its associated processor, <i>nsProc</i>		
		1	2	3
1.0	1.0	0.983417	0.983417	0.983417
0.99	0.99	0.963847	0.983027	0.983409
0.9945504 (same as in sub- section 6.2.1)	0.9986142 (same as in sub- section 6.2.1)	0.976703	0.983371	0.983416
0.999	0.999	0.981451	0.983413	0.983417
0.9999	0.9999	0.983221	0.983417	0.983417

6.2.6 Effect of Replication of Unreliable Agents (i.e. the gSAM servers) on System Availability

In this section, we replicate the gSAM servers to see their effect on system availability. In subsection 6.2.4, we have seen that the agents (i.e. the gSAM servers) in the Central subsystem are highly-sensitive components. We have replicated the gSAM server in all the three processors of the Central subsystem and observed its effect on the system availability for different number of NameServers. Figure 6.6 shows part of the MAMA model for Central subsystem group with replicated gSAM servers. Table 6.3 summarizes the results. From the table, we see that in both cases, with one or two NameServers, the replication of gSAM servers in the Central subsystem has increased the system availability by 0.554 %.

Table 6.3 System availability with or without replicated gSAM servers

Number of NameServers	System Availability without replicated gSAM servers in Central subsystem	System Availability with replicated gSAM servers in Central subsystem (as shown in Figure 6.6)
1	0.976703	0.982117
2	0.983371	0.988823

6.3 Summary

First, this chapter has demonstrated the scalability of the State Aggregation Approach for the Dependable-LQN model and the solver. It evaluated a large Air Traffic Control system which has 51 components. It implies that there are 2^{51} system states. If one were to formulate a Markov Reward model, it would have 2^{51} states similar to the Enumeration Approach. The analysis of the Fault Propagation graph in *State Aggregation Approach* did stress the storage requirements of the Aralia tool [DR97] used to determine the probability of the operational configurations, although the number of distinct operational configurations found was not excessive (just 14 in the basic case). The typical analysis time (on

Figure 6.6 Part of the MAMA model showing the interactions in the Central subsystem group. There are two replicas of gSAM servers in each processor

Windows XP hosted by Pentium (IV) processor) using the *State Aggregation Approach* was about 77 seconds (for 51 components). The time taken for solving the different variations of the model with 47 and 49 components were approximately 37 and 47 seconds respectively. Thus, although the complexity of the analysis is exponential (from the algorithms used for minpaths generation from the Knowledge Propagation graph and for operational configurations determination from the Fault Propagation graph), the time for substantial cases is not excessive. This work has been published in [DW03a].

This chapter also explored the sensitivity of system availability to the assumed failure and repair rates of components (as represented by component availabilities) in subsection

6.2.4. The results suggest that the processors and the gSAM servers (i.e. the Agents) of the Central subsystem are the most critical components. Thus, it would be better to upgrade these components than any other components to increase the system availability. As a consequence of the sensitivity analysis, subsection 6.2.6 replicated the gSAM servers of the Central subsystem and showed that the system availability did increase and thus the management architecture was improved.

Chapter 7

Model Extensions

The Dependable-LQN model described in Chapter 3 assumed constant failure rates for hardware and software components and instantaneous detection and reconfiguration. This chapter explores two possible extensions of that model. First, it describes an approach (in Section 7.1) to extend the model with aging effects. Aging changes the failure rate and performance of hardware and software components.

Second, it describes avenues (in Section 7.2) for extending the model by considering detection delay incurred due to the propagation of failure or repair information through the management architecture and the delay induced due to system reconfiguration.

7.1 Extension 1: Effect of Aging and Software Rejuvenation

This section is essentially the same as the paper [DW03c]. It describes an approach to extend the Dependable-LQN model to consider the effects of software aging and rejuvenation. The extended model is termed as *Rejuvenated-Dependable-LQN* (RDLQN) model.

The Dependable-LQN model assumed constant failures rates for the components. It would be valuable to include the effects of software aging and rejuvenation on the failures of servers, which can induce failures of applications at higher layers. Servers in this context can also include processors. For simplicity, the description here assumes perfect knowledge of the failure state. However, it can easily be extended to accommodate the role of a management architecture in limiting the knowledge of failure states (as described in Section 3.2 of Chapter 3).

Software Aging [HKKF95] is a phenomenon where a software exhibits increasing failure rate and performance degradation over time. The primary causes for this degradation are exhaustion of system resources, data corruption and accumulation of errors. Software rejuvenation [HKKF95] is a proactive technique devised for reducing the probability

of unpredicted system outages due to aging. This technique involves halting the running software, cleaning its internal state and restarting it. Examples of cleaning the internal state might be garbage collection, flushing operating system kernel tables and reinitializing internal data structures.

The RDLQN model is similar to the models proposed in [HKKF95], [GPT95], [GHKT95], [VHHT01], [CHHHTVZ01] in that it uses hypoexponentially distributed (which is an increasing failure rate distribution) time to failure for modelling aging. Generally distributed time to failure and service rate being an arbitrary function of time was considered in [GPTT98] for transaction-based systems with a single queue and [VHHT01] analyzed software rejuvenation for cluster systems. RDLQN is different from the others in the sense that it models aging in a layered system with multiple layers of queues.

The RDLQN model assumes a prediction-based rejuvenation policy as considered in [VHHT01], [CHHHTVZ01] where the rejuvenation starts whenever a degraded state of the component is detected by means of analyzing some observable symptoms. Otherwise, the component eventually goes to an undetected failed state that usually requires higher detection and repair time than the rejuvenation time.

The RDLQN model differs from the Dependable-LQN model in terms of the failure model for the individual components, which is described next.

Failure behavior of a single independent component

Figure 7.1 shows the CTMC model for a processor. Each processor initially starts in a working state (W). As time advances, it transits to a “failure probable state” (FP) with rate λ_{1p} and eventually fails (state F) with rate λ_{2p} . There is also a transition from working state (W) to completely failed state (F) with rate λ_p to model random failures caused by random events such as sudden environmental shocks etc. A failed processor can be detected and repaired with rate μ_p . Rejuvenation is not applied to processors.

Figure 7.2 shows the CTMC model for a software task, meaning an operating system process. Each task has the same model as a processor except that its degraded state (FP)

can be detected with mean detection time $1/\delta_t$ by analyzing certain symptoms. After detection, it transits to state “being rejuvenated” (BR) where the rejuvenation starts with rate μ_{2t} . Otherwise, it transits from state FP to the completely failed state (state F) with rate λ_{2t} . In case of a complete failure, the task can be repaired with rate μ_{1t} . Since rejuvenation time is usually much smaller than the repair time, thus, $\mu_{1t} \ll \mu_{2t}$.

Figure 7.1 CTMC model for each processor

Figure 7.2 CTMC model for each software task

The above CTMC's can be solved to obtain the probabilities of the component being in each of its states.

When a component is in state FP, its performance parameters are degraded but it is still operating. In state F (and BR in case of software task), the component is not operational. While a component is in one of these states, it is possible for the system to maintain operation by taking advantage of the standbys or active replicas, if any are available.

Aging of a processor is modelled by making its service rate a decreasing function of its state. Aging of a task is modelled by making the mean CPU demands of its entries an increasing function of its state.

In a Dependable-LQN model, the redundancy policies specify what needs to be done when a target server fails. However in an RDLQN model, they should additionally specify what action should be taken when a server goes to a degraded mode of operation. For example, if the primary target server is degraded, the policy might involve the client switching to a fully working standby target instead of staying with the degraded one. Similar to a Dependable-LQN model, different redundancy policies generate different operational configurations and each configuration corresponds to an LQN model and can be solved for various performance measures, for example, mean throughput, mean response time etc.

The general solution strategy for an RDLQN model is similar to that of a Dependable-LQN model except that in case of RDLQN model, it should consider multiple states for components and the redundancy policies.

Rejuvenated-Dependable-LQN Model Solution

The solution strategy of solving an RDLQN model is to:

- determine all the operational configurations considering multiple states of components and the redundancy policies.
- generate an LQN model for each configuration (taking into account that the performance parameters are different for different states of the components) and compute the performance by applying Layered Queueing Analysis,
- compute the probability of occurrence of each configuration from its component state probabilities possibly by using multi-state fault-tree analysis.
- combine the probabilities and the performance to find average performance measure for the system.

7.1.1 Example

Let us consider a client-server example (as shown in Figure 7.3) to illustrate the described approach. The operations configurations are constructed manually for this example, with

no automation. In this figure, there is one client task, Client, with entry c1. c1 makes in average, 2 requests to entry s1. The CPU demand for entries s1 and s2 are 1 and 1.5 secs respectively. The performance is degraded by 50% for both tasks Server1 and 2 in the “failure probable” state. If Server1 fails or is rejuvenating, the client uses Server2. Let us assume $1/\lambda_{1t}$, $1/\lambda_{2t}$, $1/\lambda_t$, $1/\mu_{1t}$, $1/\delta_t$ and $1/\mu_{2t}$ for both the tasks Server1 and 2 to be 10 days, 10 days, 40 days, 1 day, 10 minutes and 1 hour respectively. Also assume that all the processors (not shown here) and the client task do not age and are perfectly reliable.

Figure 7.3 A simple client-server system

There are four operational configurations for this system:

- C_1 implies Client using Server1. Server1 is in state W.
- C_2 implies Client using Server1. Server1 is in state FP.
- C_3 implies Client using Server2. Server2 is in state W.
- C_4 implies Client using Server2. Server2 is in state FP.

The probabilities of the configurations and their rewards is shown in Table 7.1 for two cases, one involving rejuvenation of both Server1 and 2 and the other without rejuvenation.

The average throughput for Client task is 0.49 requests/sec and 0.36 requests/sec with and without rejuvenation cases respectively. It is seen that rejuvenation did improve the throughput of the system significantly.

Table 7.1 Probabilities and Rewards for Operational Configurations

Configuration C_i	Probability of Configuration		Reward = Throughput of Client task
	(with rejuvenation of Server1 and 2)	(without rejuvenation)	
C_1	0.9709	0.47	0.5
C_2	0.0006	0.47	0.25
C_3	0.0275	0.0282	0.333
C_4	0.00001	0.0282	0.166
System failed	0.00099	0.0036	0

7.1.2 Explosion of Operational Configurations

One serious drawback of the model solution of an RDLQN model is the enormous number of operational configurations due to two working states of each component. This drawback could be eased by approaching the problem from different perspectives. Some of them are as follows:

- If we assume that aging only affects failure rate and degradation in performance is small enough to ignore, then both the states “W” and “FP” would have the same performance parameters. This assumption reduces the number of operational configurations by a great amount since many combinations of components states can be aggregated to one operational configuration. For this case, we can apply the methods described in [DW01a] to find the probabilities of the configurations.
- Another possibility might be to explore symmetry in redundancy. In this case, using the primary or the backup gives rise to the same performance model and thus the number of operational configurations can be reduced.
- Another way is to approximate the solution by considering only those components that affect the system in a greater amount and ignoring others. For example the components which provide shared services are usually more sensitive than others since

their failure triggers multiple failures. Identification of these components might be done by performing sensitivity analysis.

7.2 Extension 2: Effect of Delays in Detection and Reconfiguration

The down time is affected by delays in detecting and repairing the failures. These can (perhaps) be approximated by increasing the mean repair time. However a model that represents them directly would be more representative, and could evaluate architecture effects related to delay.

This section describes an approach to extend the Dependable-LQN model to consider the effects of delays incurred due to the propagation of failure or repair information through the management architecture, and system reconfiguration. The extended model is termed as *Dependable-LQN-with-Delays* (DLQND) model. Part of this approach has been published in the workshop [DW01b].

7.2.1 Expressing Detection and Reconfiguration

In order to extend the Dependable-LQN model to a DLQND model, the following additional parameters for detection and reconfiguration should be considered:

- a delay parameter for each connector in the MAMA model that represents the detection propagation delay from one task to another.
- a delay parameter for each management task in the MAMA model that represents the delay incurred due to decision-making or processing within the task.
- reconfiguration delay for each service request in the FTLQN model that has multiple targets.

Given these parameters, a directed graph can be defined that describes the sequence of detection and reconfiguration actions performed by the system in response to a failure (repair). This graph can be obtained using the FTLQN model, the MAMA model and the state of the system at the time of failure (repair). This graph is termed as the *detection-*

reconfiguration graph. Each arc in this graph represents an action and could be labeled with appropriate rates.

To demonstrate the effects of delay propagation, let us consider an example of a Dependable-LQN model shown in Figure 7.4. Figures 7.4(a) and 7.4(b) show the FTLQN model and the MAMA model augmented with the additional parameters. δ_i and $\delta_{i,j}$ (in Figure 7.4(b)) denote the delays associated with the management task i and the connector (i, j) . rd_s (in Figure 7.4(a)) denotes the reconfiguration delay for service s . A sample illustration of the detection-reconfiguration graph is provided in Figure 7.5 that defines the sequence of actions performed by the system in response to the failure of Task3 given that all the components were operational at the time of failure. The illustration in Figure 7.5 represents one detection path. This graph can also have multiple concurrent detection paths originating from the source of failure (repair). In that case, the actions along all the paths need to be completed before the system can go to the next system configuration state. Whenever multiple detection paths terminate on a node, a property on that node is specified which determines the path(s) needed to be traversed before pursuing the next action(s), if any. Figure 7.6 illustrates an example of multiple detection-paths terminating at a node, A. It describes the actions performed by the system in Figure 7.4 in response to the failure of Task 6 where all the components were operational at the time of failure. In this case, the arc emanating from node A represents a switching action of Service1 request (in Task 2) from Task 3 to Task 5. This action can only commence after the failure of both Task 3 and Task 4 has been detected by the detector, m2. This information is captured as a property of node A.

The detection-reconfiguration graph assists in conveniently visualizing the concurrent detection and reconfiguration behavior of the system. This graph is utilized to generate the set of states the system goes through in response to a failure (repair).

The runtime overhead due to sending and receiving of heartbeats in the system is modelled by adding an overhead task in each processor. An overhead task is not added in a processor where only backup tasks are running since such tasks do not contribute in

Figure 7.4(a) An FTLQN model with three layers. Alternative targets for Service1 are Tasks 3-5 in increasing order of preference

Figure 7.4(b) MAMA model for the application shown in Figure 7.4(a)

Figure 7.4 An example of a Dependable-LQN model

S1: system configuration state at the time of failure of Task 3

S2: system configuration state after the system reconfiguration is complete

Figure 7.5 Detection-reconfiguration graph for the system in Figure 7.4 in response to the failure of Task 3, where all components were operational at the time of failure

providing services. Although these backup tasks are involved in sending/receiving heartbeats, the overhead on the processor due to this activity is negligible and can be ignored. This assumption can help in reducing the number of distinct performance models.

Dependable-LQN-with-Delays Model Solution

The exact solution of the model has the following steps:

- determine all the distinct operational configurations of the system following the algorithm described in Section 4.1. In each configuration, add an overhead task on each processor representing the performance overhead due to sending and receiving of

S1: system configuration state at the time of failure of Task 6

S3: system configuration state after the system reconfiguration is complete

Figure 7.6 Detection-reconfiguration graph for the system in Figure 7.4 in response to the failure of Task 6 where all components were operational at the time of failure

heartbeats. Apply the layered queueing analysis to obtain the reward rate for each operational configuration.

- find the Markov chain that includes the reachable set of system configuration states. It incorporates the detection-reconfiguration submodel in between every two system configuration states.
- Associate the reward rate to each state of the resulting Markov chain based on the operational configuration it corresponds to.

- Solve the resulting Markov reward model to obtain the desired measures.

This exact solution suffers from the state-explosion problem. An efficient but approximated solution that provides a pessimistic bound on steady-state performability and steady-state availability could be outlined as follows:

- for each component c , estimate the maximum delay, $1/\tau_c$, spent for its failure (repair) detection and system reconfiguration. This delay can be found by summing up the delays along the longest path in the detection-reconfiguration graph that represents the sequence of actions performed by the system in response to failure (repair) of component c where all the components were operational at the time of failure.
- for each component c , create a four state Markov chain as follows: Each component starts in a working state (W). When it fails (at the rate λ_c), it transits to the state “Failed and being detected” (FBD) where detection starts at rate τ_c . After detection, it goes to the state “Failed and detected” (F) where repair commences with rate μ_c . When it is repaired, it goes to state “Repaired and being detected” (RBD) where detection starts at rate τ_c . In states FBD, F or RBD, the component is not working.

Figure 7.7 CTMC model for each component c

- Similar to the RDLQN model, generate the configurations, their rewards and their probabilities by taking into account the multiple states for components and find the performability measures.

The above solution can be approximated further by aggregating the three states (FBD, RBD and F) of Figure 7.7 to get back a two state model for each component with a slower

repair rate, which would approximate the delays within the original analysis and state space.

7.3 Summary

First, this chapter has outlined how the Dependable-LQN model can be extended to incorporate the effect of aging of components and of software rejuvenation. The main strength of this extension is its capability of capturing the aging in a system with layers of queues. One limitation might be the explosion in the number of operational configurations. This limitation can be alleviated in various ways that have been described in subsection 7.1.2. This approach has been published in the workshop [DW03c].

Second, this chapter has also outlined an approach for extending the Dependable-LQN model to consider the effects of delays in detection and reconfiguration. In this case, the exact model solution involves the generation of the Markov chain of the whole system and thereby suffers from state-explosion problem. To avoid this problem, a non-state-space oriented pessimistic bound analysis for system availability has been discussed in subsection 7.2.1. Part of this approach has been published in the workshop [DW01b].

Chapter 8

Conclusions

This research was motivated by the need for an efficient model for evaluating dependability of layered systems that employ a separate management system for performing the failure detection and reconfiguration actions on behalf of the system.

8.1 Research Summary

This thesis has developed a model (Chapter 3), called Dependable-LQN model, that includes a submodel, FTLQN, for capturing the layered service dependencies in the software architecture and a submodel, MAMA, for describing the management architecture. The model components can fail independently or they can depend on other shared failure factors. It allows a modeler to specify additional causal failure dependencies (such as AND-OR dependencies) that cannot be represented by the service-based dependencies. Four different redundancy policies are supported, namely: primary-standby, active, load-balancing and primary-standby-active redundancy. The model analysis takes into account the effect of the management architecture and the failures and repairs of the management components on the system's knowledge about the failure states of components. The model can be used for computing dependability measures, comparing various management architectures for an application (demonstrated in Section 4.3), comparing different placements of fixed number of management components (demonstrated in Section 4.4) or identifying sensitive components (demonstrated in subsection 6.2.4).

This thesis has developed an efficient solution strategy (Chapter 4), called *State Aggregation Approach (SAA)*, for the Dependable-LQN model to compute the steady-state and instantaneous measures. It avoids generating or enumerating all the system states. It uses minpath algorithms to compute the connectivity information of the MAMA model and then it generates an AND-OR-NOT graph that combines this information with the layered dependencies of FTLQN model. It then uses this graph to determine the differ-

ent operational configurations of the system (each operational configuration is a set of system states having the same performance model). For computing probabilities and performance of these configurations, it uses non-coherent fault trees and Layered Queueing Analysis respectively.

SAA has been compared with another approach called *Enumeration*. The enumeration approach involves generating all the 2^N system states (where N is the total number of components in the FTLQN and the MAMA models) while the SAA approach involves generating the operational configurations directly without enumerating the system states. In SAA approach, the number of operational configurations increases with the number of services depending on the redundancy policy of the services. If the policy is primary-backup, the increase is linear, whereas if the policy is load-balanced or active or primary-standby-active, the increase is exponential. In the worst case, the number of operational configurations can grow up to maximum $2^{M/2}$ where M is the total number of components in the FTLQN model that does not include the components in the MAMA model.

The SAA approach has been found to be computationally more efficient than the Enumeration approach for large systems. For example, in Table 4.3 (row 5), for the case of Figure 4.4 without agents, the computation times are 252 and 2.97 seconds for Enumeration and SAA approaches respectively.

A tool called Dependable-LQNS has been developed that automates both of the model solution strategies. It uses the Aralia software package for non-coherent fault trees and the Layered Queueing Network Solver (LQNS) for performance calculations for each operational configuration. There is a command-line as well as a graphical user interface for the tool and it is supported for Windows.

A large-scale Air Traffic Control system has been analyzed using the State-Aggregation approach (in Chapter 6) and it has been observed that for such a system with 51 components and 2^{51} system states, there are only 14 operational configurations and the typical analysis time was about 77 seconds. The time taken for solving the different variations of the model with 47 and 49 components were approximately 37 and 47 seconds respec-

tively. For this system, the sensitivity of system availability to the assumed component-availabilities has been studied and the management architecture was improved based on its results (in subsection 6.2.6).

Two extensions of the Dependable-LQN model have been outlined (in Chapter 7) to incorporate the effect of software aging and rejuvenation, and the delays in failure detection and reconfiguration. These extensions are straightforward and they extend the power of the model in useful ways. The solution approach for the first extension suffers from an explosion in the number of operational configurations. The second extension suffers from a state-space explosion problem. Approximations that can alleviate these problems have been outlined.

8.2 Limitations of this thesis

The Dependable-LQN model developed in this work has some limitations. First, the performance model is solved in steady state only. Steady state performability measures are then computed from steady state component availability; transient and cumulative measures for medium time-scales could in principle be found from transient component availabilities. Transient and cumulative measures at short time scales that involve short-term performance transients would require a different model.

Second, the model captures the layered service dependencies for service failure, but it assumes independent failures for the components. Some forms of dependency can be modelled by abstract "failure dependency factors" as described in subsection 3.1.2, however this does not capture all kinds of statistical dependencies.

Third, the model assumes that each component undergoes repair which is independent of other components. It can easily be extended to model repair dependencies, for example, a shared repair facility, by constructing a Markov submodel for the set of components that share a repair facility, instead of for a single component.

Fourth, the model assumes that the communication network is reliable. It can be easily extended to model network failures by introducing extra tasks for the network components

and the related dependencies.

8.3 Overall Conclusions

The techniques in this thesis provide a practical way to analyze the availability and performance properties of large software systems.

The thesis has shown that the State Aggregation Approach (SAA) can be applied successfully to substantial industrial software systems with layered architecture. Although the complexity of the computation is exponential, the solution for systems with a few dozen components have been found in minutes. The good scalability of SAA is due to aggregation of states that have the same performance model. The present limitation in size appears to originate in the Aralia tool implementation. As a result, very large systems with hundreds or thousands of components are still difficult to analyze.

Further, the framework is general enough to model the major approaches to replication [GS97] and to management architectures based on the manager-agent paradigm [SRM95].

8.4 Future Research

In this research, the computation of steady-state and instantaneous measures are considered. Computation of other measures, for example, interval measures, from the Dependable-LQN model, need further investigation.

Although the State Aggregation approach for solving the Dependable-LQN model has been found to be efficient, still better approximations could be developed. It might be challenging to decompose the system into smaller subsystems and then build the overall solution from the solution of those subsystems.

The approximations cited for the extensions considering the aging and delay effects are not implemented in this work. Implementing and analyzing those approximations need further work.

Bibliography

- [Abra79] J. A. Abraham, "An Improved Algorithm for Network Reliability", *IEEE Transactions on Reliability*, vol. R-28, April 1979, pp. 58-61.
- [AC77] A. Avizienis and L. Chen, "On the Implementation of N-version Programming for Software Fault Tolerance During Execution", in *Proc. IEEE COMPSAC 77*, Nov 1977, pp. 149-155.
- [ADM99] S. V. Amari, J. B. Dugan and R. B. Misra, "A separable method for incorporating imperfect fault-coverage into combinatorial models", *IEEE Trans. on Reliability*, 48(3), Sept 1999, pp. 267-274.
- [AK87] P. E. Amman and J. C. Knight, "Data Diversity: An Approach to Software Fault Tolerance", *Proc. 17th Intl. Symp. on Fault Tolerant Computing*, June 1987, pp. 122-126.
- [AR95] S. Asmussen and R. Y. Rubinstein, "Steady State Rare Event Simulation in Queueing Models and its Complexity Properties", in *Advances in Queueing: Theory, Methods and Open Problems*, Ed. J. H. Dshalalow, CRC Press, New York, 1995, pp. 429-462.
- [BCK98] L. Bass, P. Clements and R. Kazman, *Software Architecture in Practice*, Addison-Wesley, 1998.
- [Beau78] M. D. Beaudry, "Performance-related Reliability for Computer Systems", *IEEE Transactions on Computers*, 27(6), June 1978, pp. 540-547.
- [BG88] D. Bitton and J. Gray, "Disk shadowing", in *Proc. 14th Int. Conf. Very Large Data Base*, Los Angeles, California, 1988, pp. 331-338.
- [BGMT98] G. Bolch, S. Greiner, H. de Meer and K. S. Trivedi, *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*, John-Wiley & Sons, 1998.

- [BRJ98] G. Booch, J. Rumbaugh and I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1st edition, 1998.
- [BT86] A. Bobbio and K. S. Trivedi, "An Aggregation Technique for the Transient Analysis of Stiff Markov Chains", *IEEE Transactions on Computers*, vol. C-35, no. 9, September 1986, pp. 803-814.
- [CAS86] F. Cristian, H. Aghili and R. Strong, "Clock Synchronization in the Presence of Omissions and Performance Faults, and Processor Joins", in *16th International Conference on Fault-Tolerant Computing Systems*, 1986, pp.218-223.
- [CBCMT93] G. Ciardo, A. Blakemore, P. F. Chimento, J. K. Muppala, K. S. Trivedi, "Automated Generation and Analysis of Markov Reward Models using Stochastic Reward Nets", in *Linear Algebra, Markov Chains and Queueing Models*, C. Meyer and R. J. Plemmons (eds.), *IMA Volumes in Mathematics and Its Applications*, 48, Springer-Verlag, 1993, pp. 145-191.
- [CDD96] F. Cristian, B. Dancey and J. Dehn, "Fault-Tolerance in Air Traffic Control Systems", *ACM Transactions on Computer Systems*, 14(3), August 1996, pp.265-286.
- [CFJOQRST91] J. A. Couvillion, R. Freire, R. Johnson, W. D. Obal II, M. A. Qureshi, M. Rai, W. H. Sanders and J. E. Tvedt, "Performability Modeling with UltraSAN", *IEEE Software*, 8(5), 1991, pp. 69-80.
- [Cheu80] R. C. Cheung, "A User-Oriented Software Reliability Model", *IEEE Trans. on Software Engineering*, 6(2), 1980, pp. 118-125.
- [CHHHTVZ01] V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan and W. P. Zeggert, "Proactive Management of Software Aging", *IBM Journal of Research and Development*, 45(2), 2001, pp. 311-332.

- [CM94] O. Coudert, J.C. Madre, "MetaPrime: An interactive fault-tree analyzer", *IEEE Transactions on Reliability*, 43, March 1994, pp 121-127.
- [CMST92] G. Ciardo, R. Marie, B. Sericola and K. S. Trivedi, "Performability Analysis Using Semi-Markov Reward Processes", *IEEE Transactions on Computers*, vol. 39, no. 10, 1992, pp. 1251-1264.
- [CMT89] G. Ciardo, J. K. Muppala and K. S. Trivedi, "SPNP: Stochastic Petri Net Package", in *Proc. of the International Workshop on Petri Nets and Performance Models (PNPM)*, IEEE Computer Society Press, 1989, pp. 142-151.
- [Colb87] C. J. Colbourn, *The Combinatorics of Network Reliability*, Oxford University Press, 1987.
- [CPW89] L. N. Cassel, G. Patridge and J. Westcott, "Network management architectures and protocols: Problems and approaches", *IEEE Journal on Selected Areas in Comm.*, 7(7), Sept. 89, pp. 1104-1114.
- [Cris91] F. Cristian, "Understanding Fault-Tolerant Distributed Systems", *Communications of the ACM*, 34(2), Feb 1991, pp. 56-78.
- [CT96] T.D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems", *Journal of the ACM*, 43(2), 1996, pp. 225-267.
- [DBB92] J. B. Dugan, S. J. Bavuso and M. A. Boyd, "Dynamic fault-tree models for fault-tolerant computer systems", *IEEE Trans. on Reliability*, 41(3), Sep 1992, pp. 363-377.
- [DD95] S. A. Doyle and J. B. Dugan. "Dependability assessment using Binary Decision Diagrams", in *Proc. of the Intl. Symp. on Fault Tolerant Computing*, June 1995, pp. 249-258.
- [DDMS95] A. S. Debelack, J. D. Dehn, L. L. Muchinsky and D. M. Smith, "Next generation air traffic control automation", *IBM Systems Journal*, 34(1), 1995, pp. 63-77.

- [DDP95] S. A. Doyle, J. B. Dugan and F. A. Patterson-Hine, "A combinatorial approach to modeling imperfect coverage", *IEEE Trans. on Reliability*, 44(1), March 1995, pp. 87-94.
- [DR97] Y. Dutuit, A. Rauzy, "Exact and Truncated Computations of Prime Implicants of Coherent and non-Coherent Fault Trees within Aralia", *Reliability Engineering and System Safety*, 58, 1997, pp. 127-144.
- [DSWY91] A. Dupuy, S. Sengupta, O. Wolfson and Y. Yemini, "Design of the Net-mate network management system", *Integrated Network Management*, Elsevier Science-North Holland, 1991.
- [DW98a] O. Das and C. M. Woodside, "The Fault-tolerant layered queueing network model for performability of distributed systems", *IEEE Int. Computer Performance and Dependability Symposium (IPDS'98)*, Sept. 1998, pp. 132-141.
- [DW98b] O. Das, *Performance and dependability analysis of fault-tolerant layered distributed systems*, Master's thesis, Dept. of Systems and Computer Engineering., Carleton University, 1998.
- [DW01a] O. Das and C. M. Woodside, "Evaluating layered distributed software systems with fault-tolerant features", *Performance Evaluation*, 45 (1), 2001, pp. 57-76.
- [DW01b] O. Das and C. M. Woodside, "Failure detection and recovery modeling for multi-layered service systems", *Fifth International Workshop on Performability Modeling of Computer and Communication Systems (PMCCS-5)*, Erlangen, Germany, Sept. 2001, pp. 131-135.
- [DW02] O. Das and C. M. Woodside, "Modeling the Coverage and Effectiveness of Fault-Management Architectures in Layered Distributed Systems", *IEEE International Conference on Dependable Systems and Networks (DSN'2002)*, June 2002, pp. 745-754.

- [DW03a] O. Das and C. M. Woodside, "Layered Dependability Modeling of an Air Traffic Control System", *IEEE Intl. Conference on Software Engineering (ICSE 2003) Workshop on Software Architectures for Dependable Systems*, Portland, Oregon, USA, May 2003, pp. 50-55.
- [DW03b] O. Das and C. M. Woodside, "Dependable LQNS: A Performability Modeling Tool for Layered Systems", *IEEE International Conference on Dependable Systems and Networks (DSN 2003)*, San Francisco, California, USA, June 2003, pp. 672.
- [DW03c] O. Das and C. M. Woodside, "The Influence of Layered System Structure on Strategies for Software Rejuvenation", *Sixth Intl. Workshop on Performability Modeling of Computer and Communication Systems (PMCCS-6)*, Monticello, Illinois, USA, Sept. 2003, pp. 47-50.
- [DW04a] Olivia Das and C. M. Woodside, "Computing the Performability of Layered Distributed Systems with a Management Architecture", *ACM Fourth International Workshop on Software and Performance (WOSP 2004)*, Redwood City, California, USA, Jan 2004, 174-185.
- [DW04b] O. Das and C. M. Woodside, "Analyzing the effectiveness of fault-management architectures in layered distributed systems", *Performance Evaluation*, 56, 2004, pp. 93-120.
- [FGS98] P. Felber, R. Guerraoui and A. Schiper, "The implementation of a CORBA Object Group Service", *Theory and Practice of Object Systems*, 4(2), 1998, pp. 93-105.
- [FLP85] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of distributed consensus with one faulty process", *Journal of the ACM*, volume 32, 1985, pp. 217-246.

- [FMNPRW96] G. Franks, S. Majumdar, J. Neilson, D. Petriu, J. Rolia, and C. M. Woodside, "Performance Analysis of Distributed Server Systems," in *6th Intl. Conf. on Software Quality (6ICSQ)*, Ottawa, 1996, pp. 15-26.
- [Fran99] R. G. Franks, *Performance Analysis of Distributed Server Systems*, Ph. D. thesis, Dept. of Systems and Computer Engineering., Carleton University, 1999.
- [GCSLT86] A. Goyal, W. C. Carter, E. de Souza e Silva, S. S. Lavenberg and K. S. Trivedi, "The System AVailability Estimator", *Proc. of the 16th International Fault-Tolerant Computing Symposium*, IEEE Computer Society Press, 1986, pp. 84-89.
- [GHKT95] S. Garg, Y. Huang, C. Kintala and K. S. Trivedi, "Time and Load Based Software Rejuvenation: Policy, Evaluation and Optimality", *Proc. of First Fault Tolerant Symposium*, India, December 1995.
- [GPT95] S. Garg, A. Puliafito and K. S. Trivedi, "Analysis of Software Rejuvenation Using Markov Regenerative Stochastic Petri Net", in *Proc. of the Sixth Intl. Symposium on Software Reliability Engineering*, France, October 1995, pp. 180-187.
- [GPTT98] S. Garg, A. Puliafito, M. Telek and K. S. Trivedi, "Analysis of Preventive Maintenance in Transactions Based Software Systems", *IEEE Trans. on Computers*, 47 (1), January 1998, pp. 96-107.
- [GR93] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, San Mateo, California, 1993.
- [Gray86] J. Gray, "Why do Computers Stop and What Can be Done About it?", in *Proc. of 5th Symposium on Reliability in Distributed Software and Database Systems*, Jan 1986, pp. 3-12.
- [GS91] J. Gray and D. P. Siewiorek, "High-Availability Computer Systems", *IEEE Computer*, September 1991, pp. 39-48.

- [GS97] R. Guerraoui and A. Schiper, "Software-based replication for fault tolerance", *IEEE Computer - Special Issue on Fault Tolerance*, 30, 1997, pp. 68-74.
- [GT01] K. Goseva-Popstojanova, K. S. Trivedi, "Architecture Based Approach to Reliability Assessment of Software Systems", *Performance Evaluation*, Vol. 45/2-3, June 2001, pp. 179-204.
- [GWTH98] S. Gokhale, W. E. Wong, K. S. Trivedi and J. R. Horgan, "An Analytical Approach to Architecture-based Software Reliability Prediction", *Proc. 3rd Intl. Computer Performance and Dependability Symposium*, 1998, pp. 13-22.
- [Have95] B. R. Haverkort, "Performability modelling using DYQNTOL⁺", *International Journal of Reliability, Quality and Safety Engineering*, 1995, pp. 383-404.
- [HCKLW98] Y. Huang, P. Y. Chung, C. M. R. Kintala, D. Liang, and C. Wang, "NT-Swift: Software implemented fault-tolerance for Windows-NT", in *Proc. 2nd USENIX WindowsNT Symposium*, August 1998, pp. 47-52.
- [Heid95] P. Heidelberger, "Fast simulation of rare events in queueing and reliability models", *ACM Transactions on Modeling and Computer Simulation*, 5(1), 1995, pp. 43-85.
- [Herm90] J. Herman, "Enterprise Management vendors shoot it out", *Data Communications International*, Nov. 1990.
- [HK95] Y. Huang and C. Kintala, "Software Fault Tolerance in Application Layer", in M. Lyu (Ed.), *Software Fault Tolerance*, John Wiley & Sons, New York, 1995, Ch. 10, pp. 231-248.
- [HKKF95] Y. Huang, C. Kintala, N. Kolettis and N. D. Fulton, "Software Rejuvenation: Analysis, Module and Applications", in *Proc. 25th Annual Intl. Symp. on Fault Tolerant Computing*, California, 1995, pp. 381-390.

- [HLMR74] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith and B. Randell, "A Program Structure for Error Detection and Recovery", *Lecture Notes in Computer Science*, 16, 1974, pp. 177-193.
- [HM96] J. R. Horgan and A. P. Mathur, *Handbook of Software Reliability Engineering*, M. R. Lyu, Editor, Chapter on Software Testing and Reliability, McGraw-Hill, New York, 1996.
- [HMRT01] B. Haverkort, R. Marie, G. Rubino and K. S. Trivedi, *Performability Modelling: Techniques and Tools*, John Wiley and Sons, Chichester, England, April 2001.
- [HN89] B. R. Haverkort and I. G. Niemegeers, "Performability Modelling using Dynamic Queueing Networks", *ACM Performance Evaluation Review*, 17(1), 1989, pp. 225.
- [HN91] B. R. Haverkort and I. G. Niemegeers, "On the Mutual Performance-Dependability Influence in Dynamic Queueing Networks", in *Proc. International Workshop on Performability modelling of Computer and Communication Systems*, 1991, pp. 33-40.
- [HN96] B. R. Haverkort and I. G. Niemegeers, "Performability Modelling Tools and Techniques", *Performance Evaluation*, 25, 1996, pp. 17-40.
- [HNZ92] B. R. Haverkort, I. G. Niemegeers and P. V. van Zanten, "DyQNtool - A Performability Modelling Tool Based on the Dynamic Queueing Network Concept", in *Proc. of the 5th Intl. Conf. on Computer Performance Evaluation: Modelling Techniques and Tools*, G. Balbo and G. Serazzi (Eds.), North-Holland, 1992, pp. 181-195.
- [Howa71] R. A. Howard, *Dynamic Probabilistic Systems*, Vol II: Semi-Markov and Decision Processes, John Wiley and Sons, New York, 1971.

- [HR87] S. Hariri and C. S. Raghavendra, "SYREL: A Symbolic Reliability Algorithm based on Path Cutset Methods", *IEEE Trans. on Computers*, vol. C-36, 1987, pp. 1224-1232.
- [HT93] B. R. Haverkort and K. S. Trivedi, "Specification and Generation of Markov Reward Models", *Discrete-Event Dynamic Systems: Theory and Applications*, vol. 3, 1993, pp. 219-247.
- [Jalo94] P. Jalote, *Fault Tolerance in Distributed Systems*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.
- [KM97] S. Krishnamurthy and A. P. Mathur, "On the Estimation of Reliability of a Software System Using Reliabilities of its Components", *Proc. 8th Intl. Symp. on Software Reliability Engg.*, 1997, pp. 146-155.
- [Kreg01] H. Kreger, "Java management extensions for application management", *IBM Systems Journal*, 40(1), 2001, pp. 104-129.
- [Kuba89] P. Kubat, "Assessing Reliability of Modular Software", *Operations Research Letters*, vol. 8, 1989, pp. 35-41.
- [LABK90] J. C. Laprie, J. Arlat, C. Beounes and K. Kanoun, "Definition and Analysis of Hardware- and Software- Fault-Tolerance Architectures", *IEEE Computer*, July 1990, pp. 39 - 51.
- [Lapr84] J. C. Laprie, "Dependability Evaluation of Software Systems in Operation", *IEEE Trans. on Software Engg.*, 10(6), 1984, pp. 701-714.
- [Lapr85] J. C. Laprie, "Dependable Computing and Fault Tolerance: Concepts and Terminology", in *Proc. 15th Annual Int. Symp. on Fault-Tolerant Computing*, Ann Arbor, Michigan, June 1985, pp. 2-11.
- [Lapr92] J.C. Laprie, *Dependability: Basic Concepts and Terminology*, Dependable Computing and Fault-Tolerant Systems Series, vol.5, Springer-Verlag, 1992.

- [Lara98] L. A. Laranjeira, "NCAPS: Application high availability in UNIX computer clusters", in *Proc. 28th Annual Intl. Symposium on Fault Tolerant Computing*, Germany, June 1998, pp. 441-450.
- [LF95] A. Leinwand and K. Fang, *Network Management: A Practical Perspective*, Addison-Wesley, 1995.
- [LI95] I. Lee and R. K. Iyer, "Software Dependability in the Tandem GUARDIAN System", *IEEE Trans. on Software Engineering*, 21(5), May 1995, pp. 455-467.
- [Losq77] J. Losq, "Effects of Failures on Gracefully Degradable Systems", in *Proc. 7th Annual Int. Symp. on Fault-Tolerant Computing*, Los Angeles, CA, June 1977, pp. 29-34.
- [LTP95] D. Logothetis, K. S. Trivedi and A. Puliafito, "Markov Regenerative models", in *Int. Computer Performance and Dependability Symposium-IPDS95*, 1995.
- [Mars96] R. Marshall, *The Simple Book: An Introduction to Internet Management*, 2nd Edition, Prentice Hall, 1996.
- [Meye80] J. F. Meyer, "On Evaluating the Performability of Degradable Computing Systems", *IEEE Trans. on Computers*, 29(8), 1980, pp. 720-731.
- [MHT01] Y. Ma, J. Han and K. S. Trivedi, "Composite Performance and Availability Analysis of Wireless Communication Networks", *IEEE Transactions on Vehicular Technology*, 50(5), Sept. 2001, pp. 1216-1223.
- [MMN98] L. E. Moser, P. M. Melliar-Smith and P. Narasimhan, "A fault tolerance framework for CORBA", in *Proc. 29th Annual Int. Symposium on Fault-Tolerant Computing*, 1998, pp. 150-157.
- [MMS85] J. F. Meyer, A. Movaghar and W. H. Sanders, "Stochastic activity networks: Structure, behavior and application", *Proc. Intl. Workshop on Timed Petri Nets*, IEEE Computer Society Press, 1985, pp. 106-115.

- [MMT94] M. Malhotra, J. K. Muppala and K. S. Trivedi, "Stiffness-tolerant Methods for Transient Analysis of Stiff Markov Chains", *Microelectronics and Reliability*, 34(11), 1994, pp. 1825-1841.
- [MT91] J. K. Muppala and K. S. Trivedi, "Composite Performance and Availability Analysis using a hierarchy of Stochastic Reward Nets", in G. Balbo, ed., *Proc. Fifth International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, Torino, Italy, 1991, pp. 322-336.
- [MT92] J. K. Muppala and K. S. Trivedi, "Numerical Transient Analysis of Finite Markovian Queueing Systems", in U. N. Bhat and I. V. Basawa, Ed., *Queueing and Related Models*, Oxford University Press, 1992, pp. 262-284.
- [MW82] H. F. Martz and R. A. Waller, *Bayesian Reliability Analysis*, John Wiley and Sons, 1982.
- [Naka94] M. K. Nakayama, "Fast simulation methods for highly dependable systems", in *Proc. of 1994 Winter Simulation Conference*, IEEE Computer Society Press, 1994, pp. 221-228.
- [Nara99] P. Narasimhan, *Transparent Fault Tolerance for CORBA*, Ph.D. Dissertation, Technical Report #99-18, Department of Electrical and Computer Engineering, University of California, Santa Barbara, December 1999.
- [NF92] A. Nangia and D. Finker, "Transaction-based fault-tolerant computing in distributed systems", in *Proc. of 1992 IEEE Workshop on Fault-tolerant Parallel and Distributed Systems*, July 1992, pp. 92-97.
- [NSN99] V. F. Nicola, P. Shahabuddin and M. K. Nakayama, "Techniques for Fast Simulation of Models of Highly Dependable Systems", *IEEE Transactions on Reliability*, 50(3), 2001, pp. 246-264.

- [Prad86] D. K. Pradhan, *Fault-Tolerant Computing: Theory and Techniques*, Vol. II, Englewood Cliffs, NJ, Prentice Hall, 1986.
- [RD98] Y. Ren and J. B. Dugan, "Optimal design of reliable systems using static and dynamic fault trees", *IEEE Trans. on Reliability*, 47(3), December 1998, pp. 234-244.
- [RS95] J. A. Rolia and K. C. Sevcik, "The Method of Layers", *IEEE Trans. on Software Engineering*, 21(8), August 1995, pp. 689-700.
- [RVT95] S. Rai, M. Veeraghavan and K. S. Trivedi, "A Survey of Efficient Reliability Computation using Disjoint Products Approach", *Networks*, Vol. 25, 1995, pp. 147-163.
- [SC91] M. Sullivan and R. Chillarege, "Software Defects and Their Impact on System Availability - A Study of Field Failures in Operating Systems", in *Proc. 21st IEEE Intl. Symposium on Fault-Tolerant Computing*, 1991, pp. 2-9.
- [Schn93] F. B. Schneider, "What Good are Models and What Models are Good", in *Sape Mullender (Editor), Distributed Systems*, Addison-Wesley, 1993, pp. 17-26.
- [SFKLL98] P. Stelling, I. Foster, C. Kesselman, C. Lee and G. von Laszewski, "A fault detection service for wide area distributed computations", in *Proc. of 7th IEEE Intl. Symp. on High Performance Distributed Computing*, 1998, pp. 268-278.
- [SHL02] H. Sun, J. J. Han and I. Leventel, "Impact of Fault Management Server and Its Failure-related Parameters on High-Availability Communication Systems", *IEEE International Conference on Dependable Systems and Networks (DSN'2002)*, June 2002, pp. 679-686.
- [Shoo76] M. Shooman, "Structural Models for Software Reliability Prediction", *Proc. 2nd Intl. Conf. on Software Engg.*, 1976, pp. 268-280.

- [SKKOSS90] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel and D. C. Steere, "Coda: a Highly Available File System for a Distributed Workstation Environment", *IEEE Transactions on Computers*, Vol. 39, No. 4, April 1990, pp. 447-459.
- [SOQW95] W.H. Sanders, W.D. Obal II, M.A. Qureshi, F.K. Widjanarko, "The UltraSAN modeling environment", *Performance Evaluation*, 24(1-2), 1995, pp. 89-115.
- [SRC84] J. H. Saltzer, D. P. Reed and D. D. Clark, "End-to-end arguments in system design", *ACM Trans. on Computer Systems*, 2(4), Nov. 1984, pp. 277-288.
- [SRM95] F. Stamatelopoulos, N. Roussopoulos and B. Maglaris, "Using a DBMS for hierarchical network management", Engineer Conference, *NETWORLD+INTEROP'95*, March 1995.
- [STP96] R. Sahner, K. S. Trivedi and A. Puliafito, *Performance and Reliability Analysis of Computer Systems*, Kluwer Academic Publishers, 1996.
- [SW98] D. P. Siewiorek and R. S. Swarz, *Reliable Computer Systems: Design and Evaluation*, Chapter 7, A. K. Peters, 1998.
- [TCMS93] K. S. Trivedi, G. Ciardo, M. Malhotra and R. A. Sahner, "Dependability and Performability Analysis", in *Performance Evaluation of Computer and Communication Systems*, LNCS, L. Donatiella, R. Nelson (eds.), Springer-Verlag, 1993, pp. 587-612.
- [Tivoli] Tivoli Systems Inc., 9442 Capital of Texas Highway North, Arboretum Plaza One, Austin, Texas. See <http://www.tivoli.com>.
- [TMWH92] K. S. Trivedi, J. K. Muppala, S. P. Woolet and B. R. Haverkort, "Composite Performance and Dependability Analysis", *Performance Evaluation*, 14(3-4), 1992, pp. 197-215.

- [VHHT01] K. Vaidyanathan, R. E. Harper, S. W. Hunter and K. S. Trivedi, "Analysis and implementation of software rejuvenation in cluster systems", *ACM SIGMETRICS/Performance 2001*, pp. 62-71.
- [VT91] M. Veeraghavan and K. S. Trivedi, "An Improved Algorithm for Symbolic Reliability Analysis", *IEEE Transactions on Reliability*, R-40(3), August 1991, pp. 347-358.
- [WNPM95] C. M. Woodside, J. E. Neilson, D. C. Petriu and S. Majumdar, "The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software", *IEEE Trans. on Computers*, 44(1), January 1995, pp. 20-34.
- [Wood89] C. M. Woodside, "Throughput Calculation for Basic Stochastic Rendezvous Networks", *Performance Evaluation*, 9(2), April 1989, pp. 143-160.
- [Wood96] C. M. Woodside, "Performability modelling for multi-layered service systems", *Third International Workshop on Performability Modeling of Computer and Communication Systems (PMCCS-3)*, Illinois, USA, Sept. 1996.
- [WP00] R. B. Wallace, D. N. Prabhakar Murthy, *Reliability: Modeling, Prediction, and Optimization*, John Wiley and Sons, 2000.
- [YCA99] S. Yacoub, B. Cukic and H. Ammar, "Scenario-based Reliability Analysis of Component-based Software", *Proc. 10th Intl. Symp. on Software Reliability Engg.*, 1999, pp. 22-31.

Appendix A

BNF Description of MAMA-dl language

This section provides the formal description of the MAMA-dl language, which is used to specify a Dependable-LQN model, in BNF form. The grammar of this language is an adoption of the LQNS input file grammar provided in [Fran99]. For the nonterminals the notation *<non-terminal_id>* is used, while the terminals are written without brackets as they appear in the input text.

General Information

<inputFile> → *<generalInfo>* *<components>* *<mamaModel>* *<ftlqnModel>*

<generalInfo> → G *<comment>* *<convVal>* *<itLimit>* *<printInt>*_{opt}
*<underrelaxCoeff>*_{opt} *<endList>*

<comment> → *<string>* /* comment on the model */

<convVal> → *<real>* /* convergence value */

<itLimit> → *<integer>* /* max number of iterations */

<printInt> → *<integer>* /* intermediate result print interval */

<underrelaxCoeff> → *<real>* /* under_relaxation coefficient */

<endList> → ; /* end of list mark */

<string> → " *<text>* "

Components Information

<components> → component_description
{ *<processors>* *<tasks>* *<pseudoComponents>* }

Processor Information

$\langle processors \rangle \rightarrow \text{proc_description } \{ \langle pDeclList \rangle \}$
 $\langle pDeclList \rangle \rightarrow \langle pDeclList \rangle \langle pDecl \rangle$
 $\quad \quad \quad | /* \text{ empty } */$
 $\langle pDecl \rangle \rightarrow p \langle procId \rangle \langle procFailureRate \rangle \langle procRepairRate \rangle$
 $\quad \quad \quad \langle schedulingFlag \rangle \langle procMultiServerFlag \rangle_{opt}$
 $\quad \quad \quad \langle endList \rangle$
 $\langle procId \rangle \rightarrow \langle integer \rangle | \langle identifier \rangle /* \text{ processor identifier } */$
 $\langle procFailureRate \rangle \rightarrow \langle real \rangle /* \text{ Processor failure rate } */$
 $\langle procRepairRate \rangle \rightarrow \langle real \rangle /* \text{ Processor repair rate } */$
 $\langle schedulingFlag \rangle \rightarrow f /* \text{ First come, first served } */$
 $\quad \quad \quad | p /* \text{ Priority, preemptive } */$
 $\quad \quad \quad | h /* \text{ Head of line } */$
 $\quad \quad \quad | r /* \text{ Random } */$
 $\quad \quad \quad | s /* \text{ Processor sharing } */$
 $\langle procMultiServerFlag \rangle \rightarrow m \langle copies \rangle /* \text{ number of duplicates } */$
 $\langle copies \rangle \rightarrow \langle integer \rangle$

Task Information

$\langle tasks \rangle \rightarrow \text{task_description } \{ \langle tDeclList \rangle \}$
 $\langle tDeclList \rangle \rightarrow \langle tDeclList \rangle \langle tDecl \rangle$
 $\quad \quad \quad | /* \text{ empty } */$

$\langle tDecl \rangle \rightarrow t \langle taskId \rangle \mathbf{AT} \langle procId \rangle \langle taskFailureRate \rangle$
 $\langle taskRepairRate \rangle \langle refTaskFlag \rangle [\langle entryList \rangle]$
 $\langle taskPriority \rangle_{opt} \langle taskMultiServerFlag \rangle_{opt}$
 $\langle endList \rangle \quad /* \text{Application Task description} */$
 $| t \langle taskId \rangle \mathbf{MT} \langle procId \rangle \langle taskFailureRate \rangle$
 $\langle taskRepairRate \rangle \langle endList \rangle /* \text{Manager description} */$
 $| t \langle taskId \rangle \mathbf{AGT} \langle procId \rangle \langle taskFailureRate \rangle$
 $\langle taskRepairRate \rangle \langle endList \rangle \quad /* \text{Agent description} */$

$\langle taskId \rangle \rightarrow \langle integer \rangle | \langle identifier \rangle /* \text{task identifier} */$

$\langle taskFailureRate \rangle \rightarrow \langle real \rangle /* \text{Task failure rate} */$

$\langle taskRepairRate \rangle \rightarrow \langle real \rangle /* \text{Task repair rate} */$

$\langle refTaskFlag \rangle \rightarrow r /* \text{reference task} */$
 $| n /* \text{non-reference task} */$

$\langle entryList \rangle \rightarrow \langle entryList \rangle, \langle entryId \rangle /* \text{Entries for Applications tasks} */$
 $| \langle entryId \rangle$

$\langle entryId \rangle \rightarrow \langle integer \rangle | \langle identifier \rangle /* \text{entry identifier} */$

$\langle taskPriority \rangle \rightarrow \langle integer \rangle \quad /* \text{task priority, optional} */$

$\langle taskMultiServerFlag \rangle \rightarrow m \langle copies \rangle /* \text{number of duplicates} */$
 $| i /* \text{Infinite server} */$

PseudoComponent Information

$\langle pseudoComponents \rangle \rightarrow pseudocomp_description \{ \langle pcDeclList \rangle \}$
 $| /* \text{empty} */ \quad /* \text{No pseudoComponent block} */$

$$\begin{aligned}
\langle pcDeclList \rangle &\rightarrow \langle pcDeclList \rangle \langle pcDecl \rangle \\
&\quad | /* empty */ \\
\langle pcDecl \rangle &\rightarrow pc \langle pseudoCompId \rangle \langle pseudoCompFailureRate \rangle \\
&\quad \langle pseudoCompRepairRate \rangle \langle endList \rangle \\
\langle pseudoCompId \rangle &\rightarrow \langle integer \rangle | \langle identifier \rangle /*pseudoComponent identifier */ \\
\langle pseudoCompFailureRate \rangle &\rightarrow \langle real \rangle /* PseudoComponent failure rate */ \\
\langle pseudoCompRepairRate \rangle &\rightarrow \langle real \rangle /* PseudoComponent repair rate */
\end{aligned}$$

MAMA model interconnections

$$\langle mamaModel \rangle \rightarrow mama_model \{ \langle cDeclList \rangle \}$$

Connector Information

$$\begin{aligned}
\langle cDeclList \rangle &\rightarrow \langle cDeclList \rangle \langle cDecl \rangle \\
&\quad | /* empty */ \\
\langle cDecl \rangle &\rightarrow c \langle connectorId \rangle \langle connectorType \rangle \\
&\quad \langle fromComponent \rangle \langle toComponent \rangle \\
&\quad \langle connectorFailureRate \rangle \langle connectorRepairRate \rangle \\
&\quad \langle endList \rangle \\
\langle connectorId \rangle &\rightarrow \langle integer \rangle | \langle identifier \rangle /* connector identifier */ \\
\langle connectorType \rangle &\rightarrow AW /* Alive-watch */ \\
&\quad | SW /* Status-watch */ \\
&\quad | NTFY /* Notify */
\end{aligned}$$

$$\begin{aligned}
\langle fromComponent \rangle &\rightarrow (\mathbf{p}, \langle procId \rangle) \\
&\quad | (\mathbf{t}, \langle taskId \rangle) \\
&\quad | (\mathbf{pc}, \langle pseudoCompId \rangle) \\
\langle toComponent \rangle &\rightarrow (\mathbf{p}, \langle procId \rangle) \\
&\quad | (\mathbf{t}, \langle taskId \rangle) \\
&\quad | (\mathbf{pc}, \langle pseudoCompId \rangle) \\
\langle connectorFailureRate \rangle &\rightarrow \langle real \rangle \text{ /* Connector failure rate */} \\
\langle connectorRepairRate \rangle &\rightarrow \langle real \rangle \text{ /* Connector repair rate */}
\end{aligned}$$

FTLQN model interconnections

$$\begin{aligned}
\langle flqnModel \rangle &\rightarrow flqn_model \{ \langle entryExecutionTimes \rangle \\
&\quad \langle serviceDependencies \rangle \langle failureDependencies \rangle \}
\end{aligned}$$

Entry Execution Time Information

$$\begin{aligned}
\langle entryExecutionTimes \rangle &\rightarrow \langle entryExecutionTimes \rangle \langle entryExec \rangle \\
&\quad | \text{ /* empty */} \\
\langle entryExec \rangle &\rightarrow s \langle entryId \rangle \langle phaseServiceTimes \rangle \langle endList \rangle \\
\langle phaseServiceTimes \rangle &\rightarrow \langle real \rangle \text{ /* service time for phase 1 */} \\
&\quad | \langle real \rangle \langle real \rangle \text{ /* services times for phases 1 and 2 */} \\
&\quad | \langle real \rangle \langle real \rangle \langle real \rangle \text{ /* services times for phases 1, 2 and 3 */}
\end{aligned}$$

Service dependencies Information

```

<serviceDependencies>      → <serviceDependencies> <serviceDep>
                             | /* empty */

<serviceDep>               → y <fromEntry> <toEntry> <phaseRequests>
                             <endList> /* synchronous request */
                             | z <fromEntry> <toEntry> <phaseRequests>
                             <endList> /* asynchronous request */

                             | ALT y <fromEntry>( <serviceId>[ <entryList>])
                             <phaseRequests><endList> /*primary-standby */
                             | LB y <fromEntry>( <serviceId>[ <entryList>])
                             <phaseRequests><endList> /* load-balanced */
                             | ACTIVE y <fromEntry>( <serviceId>[ <entryList>])
                             <phaseRequests><endList> /*active */
                             | PSAR y <fromEntry>( <serviceId>[ <entryList>])
                             <endList> /* primary-standby-active */

<fromEntry>                → <entryId>

<toEntry>                  → <entryId>

<serviceId>                → <integer> | <identifier> /* service identifier */

<phaseRequests>            → <real> /*mean no. of requests in phase 1 */
                             | <real> <real> /*mean no. of requests in phase 1 and 2*/
                             | <real> <real> <real> /*mean no. of requests in phase 1, 2 and 3*/

```

Failure dependencies Information

$\langle failureDependencies \rangle \rightarrow \langle failureDependencies \rangle \langle failureDep \rangle$
 $\quad \quad \quad | /* empty */$

$\langle failureDep \rangle \rightarrow OR \langle nodeId \rangle \langle entryId \rangle [\langle entryList \rangle] \langle endList \rangle$
 $\quad \quad \quad | AND \langle nodeId \rangle \langle entryId \rangle [\langle entryList \rangle] \langle endList \rangle$
 $\quad \quad \quad | OR \langle nodeId \rangle \langle fromComponent \rangle [\langle componentList \rangle] \langle endList \rangle$
 $\quad \quad \quad | AND \langle nodeId \rangle \langle fromComponent \rangle [\langle componentList \rangle] \langle endList \rangle$

$\langle nodeId \rangle \rightarrow \langle integer \rangle | \langle identifier \rangle /* node identifier */$

$\langle componentList \rangle \rightarrow \langle componentList \rangle, \langle componentId \rangle$
 $\quad \quad \quad | \langle componentId \rangle$

$\langle componentId \rangle \rightarrow \langle integer \rangle | \langle identifier \rangle /* component identifier */$