INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Performance and Dependability Analysis of Fault-Tolerant Layered Distributed Systems

by

Olivia Das

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

Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada
April 23, 1998

© Copyright 1998, Olivia Das

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

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.

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 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.

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.

0-612-32429-X

The undersigned recommend to the Faculty of Graduate Studies
and Research acceptance of the thesis

“Performance and Dependability Analysis of Fault-Tolerant Layered Distributed
Systems”

submitted by Olivia Das, M.Sc.
in partial fulfillment of the requirements for
the degree of Master of Science

Professor C. M. Woodside, Thesis Supervisor

Chair, Department of Systems and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

April 23, 1998

Abstract

There has been extensive usage of complex, mission-critical, distributed computer systems in the recent years. Since the failure of such systems incurs high cost, there has been a great emphasis on designing these systems to be fault-tolerant. In order to achieve the correct and timely operation of such systems, one ought to model them in such a way that they meet both their performance and dependability criteria at the same time, popularly known as performability. The performability modelling has mostly been carried out using Markov reward models. The perennial problem with using the Markov Reward model is that for large systems, the state space in the Markov models becomes very large so that the generation, storage and solution of such models becomes difficult. Hence a new method has been necessary that would compute the performability of large systems without actually generating/solving their underlying Markov models. In this thesis, such a method has been introduced. The efficiency of this method lies in determining the equivalence classes of the states having the same reward (representing the performance level in a state) and their probabilities, thereby minimizing the number of performance solutions required to find the performability of a system.

Acknowledgments

I would like to express my deepest regards for my supervisor Professor C. Murray Woodside for his valuable guidance, encouragement and support. He has been a constant source of confidence and inspiration for me. I am thankful to him for introducing me to the world of Performance Engineering.

I would like to thank Carleton University, NSERC and TRIO for their financial support.

I would like to convey my regards to my parent-in-laws for their love and encouragement from India.

I would like to express my deepest love for my husband, Arindam, who has remained a constant source of inspiration, without whose encouragement and support, this thesis would have remained unfinished. Ari has always been my greatest friend I ever had. Technical discussions with him had always been a great fun. It will never be enough to thank him for whatever he did.

I convey my profound love and gratitude to my Mum-Mum who has always been there for me providing constant love and enthusiasm.

Finally, Bapi has been there always... everywhere... in everything... wherever I went, whatever I did.

I dedicate my thesis to my Bapi, Mum-Mum and Ari.

TABLE OF CONTENTS

CHAPTER 1	Introduction	1
	1.1 Motivation	1
	1.2 Layered Fault-Tolerant Systems	2
	1.3 Contributions	4
	1.4 Thesis Organization	5
 CHAPTER 2	 Background	 6
	2.1 Performability Models, Tools and Measures - A Brief Outline	6
	2.1.1 Performability Models	6
	2.1.2 Performability Modelling Tools	9
	2.1.3 Performability Measures	10
	2.2 The Layered Queueing Network (LQN) Model	11
	2.3 AND-OR Graph Concept	13
	2.4 Fault-Tree Concept	14
 CHAPTER 3	 A Formal Representation of the Fault-Tolerant Layered Queueing Network Model	 16
	3.1 The Fault-Tolerant Layered Queueing Network (FTLQN) Model Definition	16
	3.2 Fault Propagation in FTLQN Model	17
	3.3 AND-OR Graph Representation of FTLQN Model	20
	3.4 Measures of interest	22
	3.4.1 Steady-State Performability	22
	3.4.2 Point Performability	23
	3.5 Summary	23

CHAPTER 4	FTLQN Model Solution	24
4.1	The FTLQN Performability Algorithm	24
4.1.1	Approach 1: Enumeration Approach	25
4.1.2	Example	26
4.1.3	Approach 2: State Aggregation Approach	30
4.1.4	Example	36
4.2	Comparison of the Approaches: Enumeration and State Aggregation	45
4.3	Fault-Tolerant Layered Queueing Network Solver (FTLQNS)	49
4.3.1	Architecture of FTLQNS	49
4.3.2	How to use FTLQNS	50
4.4	Computation of Point Performability	53
4.5	Summary	54
 CHAPTER 5	 Application: A Call Processing System	 55
5.1	Call Processing System	55
5.2	Layered Queueing Network Model of the System	56
5.3	Fault-Tolerant Layered Queueing Network Model of the System	58
5.4	Evaluation of the State Aggregation Approach	60
5.5	Summary	68
 CHAPTER 6	 Extension of FTLQN Model for Modelling Load-Balancing Degradable Systems	 69
6.1	Extension of FTLQN Model Definition	69
6.2	Performability Algorithm for Load-Balanced FTLQN Model	72
6.3	Example	73
6.4	Summary	84

CHAPTER 7	Conclusion	85
	7.1 Research Summary	85
	7.2 A Possible Generalization of this Work	86
	7.3 Future Research	87
References		88
Appendix A	BNF Description of FTLQNS Input File Format	94
Appendix B	Sample Intermediate Outputs generated by FTLQNS	98

List of Figures

Figure 2.1	A Layered Queueing Network Model	13
Figure 2.2	Graph showing alternative subproblems for problem A	14
Figure 2.3	An AND-OR graph for Figure 2.2	14
Figure 2.4	A Fault Tree	15
Figure 3.1	A Fault-Tolerant Layered Queueing Network Model	17
Figure 3.2	Fault Propagation AND-OR graph corresponding to Fig. 3.1	21
Figure 4.1	An FTLQN Model to illustrate Enumeration Approach	26
Figure 4.2	Fault Propagation AND-OR graph corresponding to Fig. 4.1	27
Figure 4.3	LQN Models corresponding to system activation states of Fig. 4.1	29
Figure 4.4	An FTLQN Model to illustrate State Aggregation Approach	36
Figure 4.5	Fault Propagation AND-OR graph corresponding to Fig. 4.4	37
Figure 4.6	Status graphs of Fig. 4.5	38
Figure 4.7	Fault trees corresponding to the activation graphs of Fig. 4.5	40
Figure 4.8	LQN Models corresponding to system activation states of Fig. 4.4	43
Figure 4.9	FTLQN Model of a simple client-server system	45
Figure 4.10	FTLQN Model with alternative task t4 added to Fig. 4.9	45
Figure 4.11	FTLQN Model with alternative tasks t5 and t6 added to Fig. 4.10	46
Figure 4.12	FTLQN Model with tasks t7, t8 and t9 added to Fig. 4.11	46
Figure 4.13	Architecture of the FTLQNS tool	50
Figure 4.14	Flowchart of FTLQNS tool	51
Figure 4.15	Input file ex414.flqn corresponding to FTLQN model in Fig. 4.4	52
Figure 4.16	Output generated by FTLQNS for Figure 4.15	53
Figure 5.1	Call Processing System's Hardware Infrastructure	56
Figure 5.2	Layered Queueing Network Model for Call Processing System	57
Figure 5.3	FTLQN Model for Call Processing System with no stand-bys and with eight processors and failure probabilities indicated	59

Figure 5.4	FTLQN Model for Call processing System (CPS) with stand-by for Database_Manager	62
Figure 5.5	FTLQN Model for CPS with stand-bys for Database_Manager and Location_Server	63
Figure 5.6	FTLQN Model for CPS with stand-bys for Database_Manager, Location_Server and Term_Agent_Factory	64
Figure 5.7	FTLQN Model for CPS with stand-bys for Database_Manager, Location_Server, Term_Agent_Factory and Orig_Agent_Factory	65
Figure 5.8	FTLQN Model for CPS with stand-bys for Database_Manager, Location_Server, Term_Agent_Factory, Orig_Agent_Factory and ATM_NET	66
Figure 6.1	A Load-balanced FTLQN Model for a load-balancing degradable system	70
Figure 6.2	Load-Balanced Fault Propagation AND-OR graph corresponding to Fig. 6.1	71
Figure 6.3	A Load-balanced FTLQN Model to illustrate load-balanced State-Aggregation Approach	73
Figure 6.4	Load-Balanced Fault Propagation AND-OR graph corresponding to Fig. 6.3	74
Figure 6.5	Status graphs of Fig. 6.4	75
Figure 6.6	LQN Models corresponding to system activation states of Fig. 6.3	79
Figure B.1	Fault Tree generated for Figure 4.7(c)	98
Figure B.2	SHARPE's output for Figure B.1	99
Figure B.3	System Activation States and their probabilities (generated file ex414.log)	99
Figure B.4	LQNS model input file 4ex414.lqn for Figure 4.8(d)	100
Figure B.5	LQNS output file 4ex414.out for input file 4ex414.lqn in Figure B.4	101

List of Tables

Table 1	System Configuration States, their probabilities and corresponding System Operational States for Figure 4.1	28
Table 2	System Activation States and their probabilities for Figure 4.1	28
Table 3	Rewards associated with System Activation States of Table 2	29
Table 4	System Activation States and their probabilities for Figure 4.4	42
Table 5	Rewards associated with System Activation States of Table 4	44
Table 6	Execution times for System Activation States construction and their probability computation by the Enumeration Approach and the State Aggregation Approach for the Figures 4.9-4.12	48
Table 7	Execution times for System Activation States construction, their probability computation and the Model Solution by using the State Aggregation Approach for the Figures 5.3-5.8	67
Table 8	System Activation States and their probabilities for Figure 6.3	78
Table 9	Rewards associated with System Activation States of Table 8	81
Table 10	Changes in the Rewards of Table 9 due to multi-threading	83

Chapter 1 Introduction

1.1 Motivation

In recent years, there has been widespread usage of complex, mission-critical, distributed computer systems. Since the failure of such systems incurs high cost, there has been a great emphasis on designing these systems to be fault-tolerant. The fault-tolerance in these systems may be achieved by incorporating stand-by servers and a mechanism to detect a failure and to re-direct the service requests at the time of failure. Examples of such systems include telecommunication systems, aircraft control systems, banking systems, airline reservation systems and integrated office systems.

In order to achieve the correct and timely operation of the type of the systems mentioned above, it is necessary to model them in such a way that they meet their performance and dependability criteria. Performance modelling of such a system enables us to represent the probabilistic nature of the work the system is subjected to, and predict its ability to carry out work assuming that the system (or its components) does not fail. On the other hand, the dependability modelling of such a system helps us to analyze the reliability and the availability of the system based on its structure and the failure and repair behavior of its components. Since these systems may be restructured in response to faults and may continue to perform useful work, it is important to develop their models and their measures taking into account their performance and dependability simultaneously. Such a unified approach towards the modelling and measurement of the systems is popularly known as performability [MEYE80]. Performability modelling considers the effect of structural changes in response to failures and their impact on the overall performance of the system and helps us predict the performability of the system in early phases of their design.

For the performability modelling of fault-tolerant systems, the common approach has been the use of Markov reward models [MEYE82, SMIT88, TRIV92, REIB89, HAVE93a, CIAR92] in which the dependability 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 obtained by combining the reward

rates associated with the states of the Markov model with the state probabilities. For 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 by the set of states that 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 generation/enumeration of all the states of the system.

1.2 Layered Fault-Tolerant Systems

Distributed software systems are usually structured in layers, with some kind of operational control or user interface tasks as the topmost layer, making requests to various layers of servers. Client-server systems and ODP (Open Distributed Processing) systems such as DCE, ANSA and CORBA are structured this way, as are distributed processing systems. Such systems may be made fault-tolerant by incorporating stand-by servers and a mechanism to redirect the service requests to the backup server when the primary server fails.

A performability model for the above mentioned layered fault-tolerant distributed systems was first outlined by Professor C. M. Woodside in [WOOD96]. This model is here termed as Fault-tolerant Layered Queueing Network (FTLQN) model. The FTLQN model has two advantages as a style of analysis: First, it closely resembles the software architecture (making it easy to build) and second, it contains the service dependencies which are required in one form or another to analyze the failures.

This thesis presents an efficient solution algorithm for the FTLQN model in order to find its steady-state performability [MEYE80]. The steady-state performability represents the value of the reward rate offered by the system averaged over all the possible values it can provide according to the states it is in. For example, if a reward-rate equal to the system throughput is associated to each functioning state of the system, the expected value of the reward rate provides a measure of the average throughput of the system.

For the dependability part of the calculations, the solution algorithm transforms the FTLQN model into an AND-OR graph model [HORO78] (as a dependability model) in order to take advantage of the system structure and to avoid the generation and solution of the underlying Markov model. It then uses the AND-OR graph model to determine the aggregated states (each aggregated state being the set of states having the same performance model) and to obtain the probabilities of these states from the component state probabilities without generating the Markov model and without generating/enumerating all the configuration states of the system. For performance calculations, it associates a reward rate to each of these aggregated states by solving the Layered Queueing Network model [WOOD89, WOOD95a, ROLI95] (which is a performance model for client-server systems) corresponding to that state.

The solution approach presented in this thesis bears a similarity to another approach known as the Dynamic Queuing Network approach [HAVE92, HAVE89, HAVE91, HAVE90] in which a queueing network analysis is used for performance calculations and a generalized Stochastic Petri Net is used to describe the dependability aspect of the system. They are similar in the way that both of them

- first determine the possible configurations of the system by using a dependability model describing the system changes due to failure, and
- second assign a reward rate to each of those configurations by solving a performance model corresponding to that configuration.

However the work in this thesis takes advantage of the layered queueing network concept which can model the logical resources and treats the software and hardware contention uniformly, unlike the queueing network concept. This work further assumes that both the software and the hardware components can undergo failure whereas in the Dynamic Queuing Network approach, only failures of the hardware components have been considered. Besides, in this work, the dependencies are captured in the performance model whereas in the Dynamic Queuing Network approach, the dependencies are captured in the Stochastic Petri Net (the dependability model).

In the FTLQN modeling concept [WOOD96], it has been assumed that the stand-by serv-

ers are not used until needed. This thesis also extends this concept in order to model the systems involving redundant servers that can all be used at the same time with the service load being distributed equally among them. A solution algorithm has also been provided for this extended FTLQN model.

The limitation of this thesis lies in the assumption of the independent failures of the hardware and the software components in the system. This assumption is valid since a crash of a software component may be independent of the failure of its associated hardware. However, the dependencies in the system still exists through the chains of interactions exploited by the AND-OR graph model such that a failure of a hardware or software component may cause an entire distributed response to fail [WOOD96].

1.3 Contributions

The principal contributions of this thesis are as follows:

- *Development of an efficient solution algorithm for the FTLQN model:* A new solution algorithm has been developed for the FTLQN model [WOOD96] which is a performability model for fault-tolerant client-server-like distributed systems involving the stand-by server strategy where the stand-by servers are not used until needed. The algorithm can be briefly outlined as follows:
 - translation of an FTLQN model description into a specialized AND-OR graph
 - the generation of all the possible distinct aggregated states of the system (each aggregated state being the set of states having the same performance model) and the probability calculation of the system being in each of those states from the AND-OR graph *without solving the underlying Markov model of the system and without generating/enumerating all the configuration states of the system*
 - assignment of a reward rate to each of the aggregated states by applying the Layered Queueing analysis
 - the determination of the desired performability measures of the system.

The solution algorithm has been demonstrated to be efficient over another algorithm which involves the enumeration of all the configuration states of the system where the

number of such states is equal to the cardinality of the state space of the underlying Markov model of the system.

- *Development of a tool to support the technique:* A software tool termed as FTLQNS, the Fault-Tolerant Layered Queueing Network Solver, has been developed to automate the FTLQN performability modelling technique.
- *Application of the solution algorithm on a large system:* A large and complex prototype call processing system has been analyzed through the application of the solution algorithm.
- *Development of an extension to the FTLQN model* [WOOD96]: The existing FTLQN modelling concept has been extended in a way so that it can also model fault-tolerant client-server-like distributed systems involving redundant servers that can all be used at the same time with the service load being equally distributed among them. A solution algorithm has also been developed for the extended FTLQN model.

1.4 Thesis Organization

Chapter 2 provides the background material for the topics which are utilized in the development of this thesis. Chapter 3 demonstrates the formal graph-theory based representation of the FTLQN model. Chapter 4 explains two different solution algorithms of the FTLQN model, one involving the enumeration of all the configuration states of the system and the other avoiding that enumeration. It compares the efficiencies of the algorithms through some examples and also presents a high level description of the FTLQNS tool that implements both the solution algorithms. Chapter 5 presents the application of the solution algorithm that avoids enumeration, described in Chapter 4, on a prototype call processing system. Chapter 6 describes the extension of the FTLQN modelling concept [WOOD96] in order to model those fault-tolerant client-server-like distributed systems which involve redundant servers that can all be used at the same time with the service load being equally distributed among them. Chapter 7 concludes the research work and presents the potential future research directions.

Chapter 2 Background

There are four research categories that provide background to the FTLQN performability model solution. They are described in the following sections in the order: performability models, tools and measures, layered queueing network performance model, AND-OR graphs and the fault-trees.

2.1 Performability Models, Tools and Measures - A Brief Outline

Performance and dependability are two important characteristics of fault-tolerant computer systems. Usually, they are evaluated separately. In the performance analysis, it is implicitly assumed that the system (or its components) does not fail whereas the dependability analysis is based on the structure and the failure and repair behavior of the components of the system. However, when the fault-tolerant computer systems continue their operation despite the presence of component failures, their performance is usually degraded. As a consequence, for proper modelling of such systems it is important to develop measures and evaluation techniques combining performance and dependability, called performability. In this section, some recent works on performability models, tools and measures for computer systems will be reviewed.

2.1.1 Performability Models

The common approach for the development of performability models of fault-tolerant computer systems has been the use of *stochastic reward models*. In a stochastic reward model there is a *stochastic model* (also known as the structure state model) which describes the configurations of the system (i.e. structural variations of the system arising due to the failure and repair of its components) and a *reward rate* (measure of performance) which is associated with each of the states of the stochastic model. The stochastic model is thus the higher level dependability model representing the failure/repair processes of the system components. The performability measure is obtained by combining the reward rates associated with the states of the stochastic model with the state probabilities.

The research carried out so far on stochastic reward models may be broadly divided into

two following categories:

I. Markov Reward Models

In a Markov Reward Model, the underlying structure state model is a Markov model (a Continuous Time Markov Chain (CTMC)) whose dynamic behavior is such that the probability distributions for its future development depend only on the present state and not on how the process arrived in that state (Markov property). In the Markov model, the sojourn time (the amount of time in a state) is assumed to be exponentially distributed. A reward rate (measure of performance) is associated with each states of the Markov model and the desired performance measures can be obtained by combining the reward rate associated with each state of the Markov model with the state probabilities. These models are applicable to a wide range of modelling problems. Analysis of Markov Reward Models has been carried out by [MEYE82, SMIT88, TRIV92, REIB89].

The first issue related to the Markov Reward Model is concerned with the description of the dependability aspects of a system, that is, the translation of the system into its corresponding Markov model. The Markov model has some associated difficulties. First, the state space can grow much faster than the number of components in the system being modelled, making it difficult to specify a model correctly. Second, a Markov model of a system is sometimes far removed in shape and general feel from the system being modeled so that it becomes hard to translate a problem into its corresponding Markov model. These difficulties can be overcome by using a model with a form that is more concise and easy to specify. One such model is the generalized Stochastic Petri Net (GSPN) that can be utilized to formulate the Markov model automatically. In this case, the automatic generation of the Markov Reward Model is carried out by associating a reward rate with every marking of the GSPN [CIAR89, HAVE90, IBE89, MUPP91a, SAND91, CIAR92, HAVE93a, MUPP92, MUPP91b]. The use of generalized Stochastic Petri Net does not avoid the generation and solution of a large Markov state space; it merely shields the user from the largeness problem.

The next issue related to Markov Reward Model is concerned with the derivation of the reward rates. Such derivation usually require a performance evaluation in each state of the Markov model. The performance evaluation in each state of the Markov model may be measurement based or model based. In the latter case a simple combinatorial performance model

has been used [BLAK88, SMIT88, SMIT89], a queueing network model has been used [LOPE93, TRIV90, SMIT89, HAVE89] and a Markov model has been used [MUPP91a]. The performance index chosen as reward rates may be any measure such as throughput, buffer loss probability, response-time deadline violation probability and so forth [MUPP92, MUPP91b, MUPP91a, TRIV90].

A version of the Markov Reward Modelling approach known as Dynamic Queueing Network concept has been developed [HAVE92, HAVE89, HAVE91, HAVE90] in which a parameterized queueing network is used to model the performance aspect of the system and a generalized stochastic Petri net is used to describe the dependability aspect of the system. This technique comprises of a function that maps the possible markings of the generalized stochastic Petri net, denoting the system configurations, to the queueing network parameters. Then, a reward rate is associated with every possible generalized stochastic Petri net markings by solving the corresponding queueing network models. Finally, the actual calculations for performability are done by translating the dynamic queueing network model to a Markov Reward model. This work has considered the problem of minimizing the number of performance solutions required. It groups the markings corresponding to the same queueing network parameters by enumerating them and solves the queueing network model once for each group. This work 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, WOOD95a, ROLI95] which can model the logical resources and treats the software and hardware contention uniformly, unlike the queueing network concept. It also assumes that both the software and the hardware components can undergo failure, whereas in the Dynamic Queueing Network approach only the failure of the hardware components have been considered.

In a Markov Reward Model, 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 value of the reward rate. 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.

II. Non-Markovian Reward Models

In a Markov Reward Model, the amount of time in a state is assumed to be exponentially distributed. There are a great number of real situations in which the failure processes are non-poisson and the failure times of the components are deterministic or generally distributed. For dealing with non-exponentially distributed failure times, the semi-Markov Reward Model has been developed. In a semi-Markov Reward Model, the underlying structure state model is a semi-Markov process (SMP) in which the rate of transition from state i to j may depend on how long the chain has been in state i , but it still does not depend on anything that happened before the chain reached state i . In SMP, the amount of time in a state is allowed to be any distribution function. Analysis of semi-Markov Reward Model has been carried out by [CIAR90]. There are other complex non-Markovian performability models like Markov Regenerative Reward Model [LOGO95a, LOGO95b, GARG95] which have not been surveyed here. Overall, the analysis of non-Markovian models are rather difficult because of their high complexity.

2.1.2 Performability Modelling Tools

In order to automate the performability modelling techniques, various tools have been developed in the recent years. Among these tools, Numas [CLOS95], Metasan [SAND85, SAND87, SAND91], Metfac [CARR86], SAVE [GOYA86, GOYA87a, GOYA87b, GOYA87c], SPNP [CIAR89, CIAR91], Tangram [PAGE89], Penpet [LEPO91], UltraSAN [COUV91, OBAL94], DqnTool [HAVE92, HAVE89, HAVE91, HAVE90, HAVE95], Surf-2 [BEOU93] are based on the Markov Reward Model whereas Metaphor [FURC84] is based on acyclic semi-Markov Reward model. The tool SHARPE [SAHN96, SAHN87a, SAHN87b, SAHN86, SAHN93] allows for the construction of both Markov and semi-Markov Reward models. SHARPE can also be used to construct and analyze reliability models. These tools have been surveyed in [HAVE96, HAVE90].

The common limitation related to all the above tools is that they do not consider the generation of the equivalence classes of states, each class being the set of states having the same

performance model, directly without solving the underlying Markov model or without generating or enumerating all the states of the system. Moreover, in case of some of the tools such as Metfac, Tangram and SHARPE, the rewards have to be supplied manually.

2.1.3 Performability Measures

The goal of evaluating the performability of a system is to capture the overall quality of its responsiveness which is sometimes degraded by failures. Performability was first introduced by [MEYE80] as a measure that quantifies a system's ability to perform in the presence of faults. Such ability has been formally defined as the probability that the system will perform above a given accomplishment level, given that the system has been operational for a period of time t . This section defines various performability measures of interest as described in [HAVE96, SAHN96].

Let χ denote the set of all possible configurations in which the system can operate. Let us consider R to be a real-valued reward rate function on the state space χ . The reward rates may be any performance measure such as throughput, buffer loss probability, response-time deadline violation probability and so forth [MUPP92, MUPP91b, MUPP91a, TRIV90]. Let the row vector $p = [..., p_i, ...]$ denote the steady-state probability distribution of states in χ and $p(t) = [..., p_i(t), ...]$ be the vector of state probabilities at time t . Let $Z(t)$ be the system reward rate at time t i.e. the reward value of the state of the system at time t . Then the following measures of interest can be defined:

- (1) the steady-state performability or the expected steady-state reward rate is given by,

$$P_{steady} = E[Z] = \sum_{i \in \chi} p_i R(i)$$

This quantity represents the value of the reward rate offered by the system averaged over all the possible values it can provide according to the states it is in.

We can partition the set of states χ in a set χ_u of “up” states, and a set χ_d of “down” states, i.e. $\chi = \chi_u \cup \chi_d$, with $\chi_u = \{i \in \chi \mid R(i) > 0\}$ and $\chi_d = \{i \in \chi \mid R(i) = 0\}$. Consequently, we then have,

$$P_{steady} = \sum_{i \in \chi} p_i R(i) = \sum_{i \in \chi_u} p_i R(i)$$

In a particular case, when all the reward rates $R(i) = 1$, whenever $x \in \chi_u$ and $R(i) = 0$ else-

where, then the above equation gives the steady-state dependability D as:

$$D = \sum_{i \in \chi_u} p_i$$

(2) the point performability or the expected instantaneous reward rate is given by,

$$P_{instant}(t) = E[Z(t)] = \sum_{i \in \chi} p_i(t)R(i)$$

Whenever $R(i) = 1$ for all the system operational states and zero otherwise, the above equation gives the system instantaneous dependability.

(3) the cumulative performability or the accumulated reward in $[0, t)$ which represents the amount of work accomplished during a generic time interval $[0, t)$ is given by,

$$Y(t) = \int_0^t Z(\tau) d\tau$$

(4) the performability distribution or the distribution of the accumulated reward is given by,

$$F(t, y) = Prob\{Y(t) \leq y\}$$

(5) the mean accumulated reward in $[0, t)$ is given by,

$$E[Y(t)] = E\left[\int_0^t Z(\tau) d\tau\right] = \int_0^t E[Z(\tau)] d\tau = \sum_{i \in \chi} R(i) \int_0^t p_i(\tau) d\tau$$

The fraction $E[Y(t)]/t$ is known as the interval availability when a reward rate 1 is assigned to the system operational states and zero to non-operational states.

2.2 The Layered Queueing Network (LQN) Model

A Layered Queueing Network (LQN) model [WOOD95a, WOOD89, ROLI95] is a performance modelling framework for client-server-like distributed systems that use a style of synchronous inter-task communication, such as remote procedure calls (RPCs), where the sender is blocked until the receiver replies. The framework can also accommodate asynchronous interactions (with no reply), but they are not considered in our work. An LQN model can be solved for performance metrics by the tool LQNS (Layered Queueing Network Solver) [FRAN95, WOOD95b].

An LQN model describes objects called “tasks” which may represent software processes (clients and servers) or hardware devices (e.g. processors, disks). Tasks may send or receive

service requests. Each of these service requests is handled by a service handler, known as an “entry”, embedded in the task accepting the request. One task may have one or more entries providing various services. In object-oriented terms a task is an object and an entry is a method. Every software task is allocated to a processor which is also a server. An entry may have a first phase of service before sending the reply, and an optional second phase 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 phase the server has an independent thread of control of its own. A task which originates requests, such as a user interface task or a system control task, is called a “reference task”. It has only a sequence of second phases, as it accepts no requests. In order to avoid circular chains of blocked tasks causing deadlocks, the attention is restricted to acyclic models.

Figure 2.1 shows a Layered Queueing Network model with one reference task (Task 1), four “active servers” (Tasks 2, 3, 4, 5) (which also makes requests), and three pure servers (Processors 1, 2 and 3). In this figure, the numbers in square brackets are the execution times of each entry, by phase, while those in round brackets are the mean numbers of messages from entry to entry, by the phase of the sender.

The present research uses the LQN performance model for the reward rate derivations.

Figure 2.1 A Layered Queueing Network Model

2.3 AND-OR Graph Concept

AND-OR graphs [HORO78] are used in Artificial Intelligence for the purpose of solving a problem by reducing it to a set of alternative subproblems. The graph of Figure 2.2 represents a problem A that can be solved by either solving both the subproblems B and C or one of the subproblems D or E. A group of subproblems that must be solved in order to imply a solution to the parent node is indicated by an arc joining the respective edges. By introducing dummy nodes F and G, as in Figure 2.3, all nodes can be made to be such that their solution requires either all descendents to be solved or only one descendents to be solved. Nodes of the first type are known as AND nodes and those of the latter type OR nodes. Nodes A and G in Figure 2.3 are OR nodes while node F is an AND node. AND nodes are usually drawn with an arc across all edges leaving the node. Graphs like Figure 2.3 are called AND-OR graphs.

Figure 2.2 Graph showing alternative subproblems for problem A

Figure 2.3 An AND-OR graph for Figure 2.2

The AND-OR graphs have been previously used in [SAHN87b] for dependability purposes. The present research also uses AND-OR graphs for representing the dependability aspects of a system.

2.4 Fault-Tree Concept

The fault tree [SAHN96] is a pictorial representation of the combination of *conditions* that can cause the occurrence of system failure. A *condition* at level i is reduced to a combination of lower-level *conditions* by means of logic gates. The process of reduction stops when the *basic conditions* are reached where a *basic condition* is a condition that cannot be reduced further. The component failures can be the *basic conditions*. It is assumed that the *basic conditions* are mutually independent and that their probabilities are known. The *condition* is denoted by a value of logic 1 for failure at a node; otherwise the logic value of the node is 0. Each gate has inputs and outputs. The input to a gate is either a *basic condition* or the output of another gate. The output of an *and* gate is a logic 1 if and only if all of its inputs are logic 1. The output of an *or* gate is a logic 1 if and only if one or more of its inputs are at logic 1. The output of a *k out of n* gate is a logic 1 if k or more of the inputs are at logic 1. There is a single output called the *top condition* representing the occurrence of system failure.

Let us consider an example of a fault-tolerant computer system with two processors and three memory modules. Assume that one of the memory modules (M3) is shared and the other two are private, one for each processor. The system will operate as long as there is at least one operational processor with access to either a private or shared memory. This system can be modeled by a fault-tree as shown in Figure 2.4. The structure tells us when the system has failed, i.e. when the *top condition* is a value of logic 1.

Figure 2.4 A Fault Tree

A fault-tree with or without repeated *conditions* can be analyzed by using the Sum-Of-Disjoint-Products algorithm [RAI95, VEER91, ABRA79, HARI87]. The fault-tree calculation can be achieved using the tool SHARPE [SAHN96, SAHN87a, SAHN87b] that uses this Sum-Of-Disjoint-Products algorithm.

Chapter 3 A Formal Representation of the Fault-Tolerant Layered Queuing Network Model

A high-level sketch of the FTLQN performability model was provided in [WOOD96]. This chapter introduces a formal mathematical representation of the FTLQN performability model. This chapter also provides the performability measures of interest for the FTLQN model.

The FTLQN performability modelling concept can be applied to model a class of systems which possess a fault-tolerant client-server-like structure and whose design takes advantage of the stand-by redundancy approach, an approach that advocates the usage of a stand-by server when the primary server fails and a mechanism to re-direct the service requests at the time of failure.

Since AND-OR graphs [HORO78] may provide a convenient means to model the dependencies of the service failures in a layered system, the solution algorithm for the FTLQN model described in next chapter proposes to use an AND-OR graph model for the dependability part of its calculations. Hence it is necessary to transform the FTLQN model description into a formal AND-OR graph representation.

3.1 The Fault-Tolerant Layered Queueing Network (FTLQN) Model Definition

An FTLQN model [WOOD96] describes distributed systems which employ alternative servers and routing of requests to mask failures. It modifies the LQN model to express the strategy to be used in case of failure and to generate the different configurations in which the system may be fully or partly operational. The layered structure of an FTLQN model describes the dependencies that determine the service failures, based on service dependencies.

Figure 3.1 A Fault-Tolerant Layered Queueing Network model

In an FTLQN model a service request arc may be replaced by a set of alternative arcs with an order of preference. Figure 3.1 shows the same system as Figure 2.1 in Chapter 2, with Task 4 (now named Task 4a) having a stand-by server, Task 4b. Such a set of arcs is shown attached to a solid black rectangle on Entry 2, with the order of preference indicated by #1, #2. That is if possible, Entry 2 gets this service from Entry 4a, but if Entry 4a is unavailable, it will switch to Entry 4b.

The next sections describe in order: how failures are propagated from lower layers to upper layers in an FTLQN model, the AND-OR graph-based representation of an FTLQN model, and the measures that can be associated with the FTLQN model.

3.2 Fault Propagation in FTLQN Model

In an FTLQN model, it is assumed that any of the entities can undergo a failure, which is independent of the failure of other entities in the model (where an entity refers to a software task or a processor).

Definition 3.1 The *component state* s_i of an entity i , is defined to be either 0 or 1 corresponding to its failed state or working state.

Each entity i is governed by a separate Markov chain with a working state ($s_i = 1$) and a failed state ($s_i = 0$), with failure rate λ and repair rate μ which gives its steady-state probabilities $Prob(s_i = 0)$ and $Prob(s_i = 1)$ as:

$$Prob(s_i = 0) = \lambda / (\mu + \lambda) \text{ and}$$

$$Prob(s_i = 1) = \mu / (\mu + \lambda).$$

Let us consider that there are N entities in the system.

Definition 3.2 The *system configuration state* S is defined to be a vector of *component states* of entities, given by $S = \{s_1, s_2, \dots, s_N\}$.

Because of the layered structure of an FTLQN model, the failure of a service provided by an entry depends on the failure of the services provided by its processor, its task and by other lower-level entries in the model. We define an entry e of task t to be *working* if

- the task t is working (i.e. $s_t = 1$),
- and the processor p allocated to task t is working (i.e. $s_p = 1$),
- and all the services that the entry e uses are working. A service used by an entry is said to be working if any of the alternative entries providing that service is working.

The entry e is failed otherwise. The dependency on t and p are represented separately because a crash of task t may be independent of the failure of processor p .

Definition 3.3 An *active entry* is defined to be a working entry which is being used for service by the system.

Let us consider that there are E entries in the FTLQN model.

Definition 3.4 The *operational state* s_e^* of an entry e is defined to be either 0 or 1 corresponding to its failed or working state as defined above.

Definition 3.5 The *system operational state* S^* is defined to be a vector of *operational states* of entries given by $S^* = \{s_1^*, s_2^*, \dots, s_E^*\}$.

Because of the dependencies in the layered structure of the FTLQN model, many *system*

configuration states would correspond to a single *system operational state*.

Definition 3.6 $f^*: \{S\} \rightarrow \{S^*\}$ is a function that maps a *system configuration state* S to a *system operational state* S^* . The function f^* is a surjection since many *system configuration states* may correspond to a single *system operational state* because of the propagation of dependencies in the layered structure of FTLQN model.

Definition 3.7 A *system activation state* S^{**} is an aggregated state that is defined to be a set of *system operational states* which have the same *active entries* and thus have the same performance model and the same reward. A state S^{**} includes “*don't cares*” for the entries whose status does not affect the choice of *active entries*. Each S^{**} can therefore be concluded to have a distinct performance model.

From definition 3.6 and 3.7, it follows that a single *system activation state* would also correspond to a set of *system configuration states*.

Definition 3.8 $g^*: \{S^*\} \rightarrow \{S^{**}\}$ is a function that maps a *system operational state* S^* to a *system activation state* S^{**} as per definition 3.7. The function g^* is a surjection since many *system operational states* may correspond to a single *system activation state* as per definition 3.7 above.

From the definition 3.6, 3.7 and 3.8, a *system activation state* S^{**} can be expressed as:

$$\begin{aligned} S^{**} &= \{S^* \mid g^*(S^*) = S^{**}\} \\ &= \{S \mid f^*(S) = S^* \text{ and } g^*(S^*) = S^{**}\} = \{S \mid g^*(f^*(S)) = S^{**}\} \end{aligned}$$

Definition 3.9 Define an equivalence relation eq on the set of *system configuration states* as:

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

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

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 *system activation state*. The states contained in a *system activation state* S^{**} correspond to the same performance model.

The dependencies among the *operational states* of the entries can be computed using an AND-OR graph. In the use of AND-OR graphs in this work, the meaning of a node being

“true” is “service is active”, rather than the “subproblem is solved”. The next section describes the graph representation of an FTLQN model. It is to be noted that in an AND-OR graph, a node being “true” implies “service is active” whereas in a fault tree, a node being “true” implies “service is failed”.

3.3 AND-OR Graph Representation of FTLQN Model

The fault dependencies in an FTLQN model can be represented by a prioritized, labelled and directed AND-OR graph termed as a *fault-propagation AND-OR graph*, which is defined as a six-tuple $G = (V, E, r, L, priority, \Pi)$ where

1) V is the set of nodes in the graph G . V contains four types of nodes, namely, *entry* node, *service* node, *processor* node and *task* node which are defined as follows:

(i) an *entry* node is an AND node that describes an entry of the FTLQN model. The set of entry nodes is denoted by V_{entry}

(ii) a *service* node is an OR node that describes the preference order of the alternative targets for a requested service. All of its successors represent the alternative targets while its parent represents the entry requesting the service. This node corresponds to the solid black rectangle in the FTLQN model and is represented as a shaded node in G . The set of *service* nodes is denoted by $V_{service}$

(iii) a *processor* node is a leaf node that contains the information about a processor of the FTLQN model. The set of processor nodes is denoted by V_{proc}

(iv) a *task* node is a leaf node that contains the information about a software task of the FTLQN model. The set of task nodes is denoted by V_{task}

V also has one special start node r which is an OR node representing the overall state of the system. Its successors represent the reference entries of the FTLQN model, a reference entry being an entry of a reference task.

2) E is a subset of $V \times V$, the set of edges.

3) $L = V_{proc} \cup V_{task}$ is the set of leaf nodes, which can fail independently.

4) *priority* is a function $priority: E \rightarrow \mathcal{W}$ that labels a whole number $priority(e)$ to each

edge $e = (v, w) \in E$, where $v, w \in V$. If $v \in V_{entry}$, then $priority(e) = 0$ so that all the edges outgoing from an *entry* node would have equal priorities. If $v \in V_{service}$, then $priority(e)$ is the preference order of the target w (lower value for higher preference). Also if $v = r$, then $priority(e) = 0$ so that all the edges outgoing from r have equal priorities.

5) Π is a function $\Pi: L \rightarrow \{\pi | (\pi \in R) \wedge (0 \leq \pi \leq 1)\}$ that assigns a real number $\Pi(l)$ to each leaf node $l \in L$, representing the probability of node l being in the “working” state. These leaf nodes represent entities that may fail independently.

Figure 3.2 shows the *fault propagation AND-OR graph* corresponding to the FTLQN model in Figure 3.1 of Section 3.1. An edge e is labelled with $\#priority(e)$ in Figure 3.2. An edge with priority 0 is not labelled. The node Service-4 is a *service* node that describes the preference order of the alternative targets Entry 4a and 4b for the service requested by Entry 2 and corresponds to the solid black rectangle on Entry 2.

Figure 3.2 Fault Propagation AND-OR graph corresponding to Fig. 3.1

The *fault propagation AND-OR graph* is used in the solution algorithm, described in the next chapter, to generate the distinct *system activation states* and the probabilities of the system being in those states.

3.4 Measures of interest

An FTLQN model can yield two types of measure both at steady state and in transient conditions:

- steady-state performability or expected value of reward rate
- point performability or expected instantaneous reward rate

Many works on performability stress the importance of transient measures as opposed to the steady-state measure. Steady-state measures are however useful for a number of reasons provided in [HAVE93b] as follows:

- For highly dependable systems such as telephone and telecommunication switches, steady-state performability measures are often part of the requirement specifications, so that calculation of them is important.
- Under natural monotonicity conditions, the steady-state performability provides a lower bound for the point performability.

For the above reasons, in this work, the evaluation of the steady-state performability has been emphasized.

The following sub-sections introduce and define the measures of interest for an FTLQN model.

3.4.1 Steady-State Performability

Steady-State Performability represents the value of the reward rate offered by the system averaged over all the possible values it can provide according to the states it is in. From a mathematical point of view, this quantity can be obtained as follows:

Let S_i denote a *system configuration state* and R_i be the reward associated with the state S_i . Then,

$$P_{steady} = \sum_i R_i Prob(S_i)$$

Let the set of all *system configuration states* be partitioned by the equivalence relation *eq* into n equivalence classes (all the *system configuration states* in each class have the same performance model and hence have the same reward) represented by the *system activation states* $S_1^{**}, S_2^{**}, \dots, S_n^{**}$. Let $Prob(S_j^{**})$ denote the steady-state probability of the system being in state S_j^{**} . Then, P_{steady} can be expressed as:

$$P_{steady} = \sum_{j=1}^n R_j Prob(S_j^{**}) \text{ where } Prob(S_j^{**}) = \sum_{S \in S_j^{**}} Prob(S)$$

If a reward rate R_j is taken to be the system throughput, then the steady-state performability P_{steady} would be the mean throughput reward.

3.4.2 Point Performability

The point performability $P_{instant}(t)$ of the system is defined as the expected instantaneous reward rate at a generic time t and is given by:

$$P_{instant}(t) = \sum_{j=1}^n R_j Prob_t(S_j^{**}), \text{ where } Prob_t(S_j^{**}) \text{ represents } Prob(S_j^{**}) \text{ at time } t.$$

3.5 Summary

This chapter has introduced a formal description of the FTLQN performability model. It has presented the relevant definitions that will be used in the subsequent parts of this thesis. It has demonstrated how an FTLQN model can be transformed into its corresponding *fault propagation AND-OR graph* exploiting the dependencies among the services in the model. It has further introduced the measures of interest for the FTLQN model.

Chapter 4 FTLQN Model Solution

This chapter introduces two different solution strategies for the FTLQN model[WOOD96] in order to compute its steady-state performability. Both solution strategies use the AND-OR graph representation of the FTLQN model for the dependability calculations in order to avoid solving the underlying Markov model and use the Layered Queueing Network analysis for the performance calculations. The first strategy involves the enumeration of all the *system configuration states* (the number of *system configuration states* being equal to the number of states in the underlying Markov model) whereas the second strategy avoids such enumeration and directly finds the equivalence classes of *system configuration states* having the same performance model. This chapter also presents a high level description of the FTLQNS tool that implements both the solution strategies. It then provides some examples in order to demonstrate the efficiency of the second strategy over the first one for the scaled systems using the FTLQNS tool. It further discusses the computation of point performability of an FTLQN model.

4.1 The FTLQN Performability Algorithm

This section describes an algorithm to calculate the steady-state performability of an FTLQN model as follows:

- 1) Obtain the *fault propagation AND-OR graph*, G , corresponding to the FTLQN model, as described in Chapter 3.
- 2) Determine all the possible distinct *system activation states*, $S_1^{**}, S_2^{**}, \dots, S_n^{**}$, from G and calculate the steady-state probability, $Prob(S_i^{**})$, of the system being in state S_i^{**} for $i = 1, 2, \dots, n$ by one of two approaches described later in subsections 4.1.1 and 4.1.3.
- 3) Generate an LQN model for each S_i^{**} and apply the Layered Queueing Network Analysis to obtain its performance measures. Select a performance measure, R_i , and assign it as a reward to S_i^{**} .

4) Compute the steady-state performability, P_{steady} , of the system given by

$$P_{steady} = \sum_{i=1}^n R_i Prob(S_i^{**}).$$

Next, the step 2 of the above algorithm is elaborated. One way to achieve this step involves the enumeration of all the $2^{|L|}$ *system configuration states* as described below in approach 1. A better way is attempted under approach 2 which avoids such enumeration.

4.1.1 Approach 1: Enumeration Approach

This approach creates all the possible distinct *system activation states* from the corresponding *fault propagation AND-OR graph*, G , and also computes the probability of the system being in each of those states by enumerating all the $2^{|L|}$ *system configuration states* where $|L|$ is the number of leaf nodes in graph G representing hardware components and the software tasks that may fail independently.

Let $l_1, l_2, \dots, l_{|L|}$ denote the $|L|$ entities in the system represented by the leaf nodes in graph, G . If one were to generate the $2^{|L|}$ *system configuration states*, then the set of all *system configuration states* could be partitioned by the relation *eq* into equivalence classes, each class being represented by a single *system activation state*.

Let $A = \{(a, Prob(a)) \mid a \text{ is a system activation state}\}$.

(a) Initialize A to be empty.

(b) For each *system configuration state* $S_j = \{s_1, s_2, \dots, s_{|L|}\}$, where $j = 1, 2, \dots, 2^{|L|}$ and $s_i = 0$ or 1,

$$\bullet \text{ compute } Prob(S_j) = \prod_{i=1}^{|L|} Prob(l_i \text{ being in state } s_i),$$

$$\text{where } Prob(l_i \text{ being in state } s_i) = \Pi(l_i) \text{ if } s_i = 1$$

$$= 1 - \Pi(l_i) \text{ if } s_i = 0$$

$\bullet$ find the corresponding *system operational state* $f^*(S_j)$ by traversing G in depth-first-search fashion[NILS80] taking into account the priorities associated with the edges

- if $\exists (a, Prob(a)) \in A$ such that $g^*(f^*(S_j)) = a$, then $Prob(a)$ becomes $Prob(a) + Prob(S_j)$ and S_j is included in the equivalence class represented by the *system activation state* a
- else A becomes $A \cup \{(a', Prob(a'))\}$ where $a' = \{S_j\}$ and $Prob(a') = Prob(S_j)$.

The following example illustrates the use of this *enumeration approach*.

4.1.2 Example

To see how the enumeration algorithm works, let us consider an FTLQN model with one reference task as shown in Figure 4.1. In this figure, there are three tasks t1-t3 with entries e1-e3 sharing the two processors, p1 and p2. Suppose the independent failure probabilities for (p1, t1, p2, t2, t3) are in order (0.05, 0.1, 0.07, 0.13, 0.16).

Figure 4.1 An FTLQN Model to illustrate Enumeration Approach

In this example, there are $2^5 = 32$ *system configuration states* represented by $S_j = \{s_{p1}, s_{t1}, s_{p2}, s_{t2}, s_{t3}\}, j = 1, 2, \dots, 32$. The FTLQN model in Figure 4.1 is transformed into its corresponding *fault propagation AND-OR graph* as shown in Figure 4.2. In Figure 4.2, serv1 is the service node corresponding to the solid black rectangle on entry e1 in Figure 4.1 representing the service requested by entry e1.

Figure 4.2 Fault Propagation AND-OR graph corresponding to Fig. 4.1

All the *system configuration states* where the processor p1 has failed would be included in the system failed state. There are 16 such states. Again, out of the remaining $(32 - 16 = 16)$ states, there are 8 states where the task t1 has failed and which also causes the system to fail. Thus, there are 8 remaining *system configuration states* where both p1 and t1 are working. Out of these 8 states, there are 3 states where p2 or t2 or both have failed and t3 has failed so that these 3 states would be included in the system failed state after the graph traversal. Thus, the system failed state has probability equal to the sum of probabilities of these $16 + 8 + 3 = 27$ states. The remaining 5 *system configuration states* that may correspond to an operational configuration of the system are provided in Table 1 with their probabilities. The corresponding *system operational states* found after the graph traversals are also shown in this table. This table does not show the other 27 *system configuration states* and their corresponding *system operational states* separately. However, it shows the sum of their probabilities.

Table 1: System Configuration States, their probabilities and corresponding System Operational States for Figure 4.1

j	System Configuration State $S_j = \{s_{p1}, s_{t1}, s_{p2}, s_{t2}, s_{t3}\}$	System Operational State $f^*(S_j) = \{s_{e1}^*, s_{e2}^*, s_{e3}^*\}$	$Prob(S_j)$
1	[1, 1, 0, 0, 1]	[1, 0, 1], e1 and e3 are <i>active entries</i>	0.00654
2	[1, 1, 0, 1, 1]	[1, 0, 1], e1 and e3 are <i>active entries</i>	0.04374
3	[1, 1, 1, 0, 1]	[1, 0, 1], e1 and e3 are <i>active entries</i>	0.08683
4	[1, 1, 1, 1, 0]	[1, 1, 0], e1 and e2 are <i>active entries</i>	0.1107
5	[1, 1, 1, 1, 1]	[1, 1, 1], e1 and e2 are <i>active entries</i>	0.5811
Other 27 System Configuration States		System Operational States with entry e1 <i>inactive</i>	Σ probabilities of these 27 states = 0.1711

When the active entries in the middle column in Table 1 are examined, it is found that $g^*(f^*(S_1)) = g^*(f^*(S_2)) = g^*(f^*(S_3))$, i.e. $S_1 \text{ eq } S_2 \text{ eq } S_3$ and $g^*(f^*(S_4)) = g^*(f^*(S_5))$, i.e. $S_4 \text{ eq } S_5$. The other 27 system configuration states corresponds to the system failed state. Thus there are three equivalence classes represented by the three system activation states as shown in Table 2.

Table 2: System Activation States and their probabilities for Figure 4.1

j	System Activation State S_j^{**}	$Prob(S_j^{**})$
1	$\{S_1, S_2, S_3\}$	$0.00654 + 0.04374 + 0.08683 = 0.1371$
2	$\{S_4, S_5\}$	$0.1107 + 0.5811 = 0.6918$
3	System Failed State	0.1711

An LQN model for each of the system activation states S_1^{**} and S_2^{**} is generated as shown in Figure 4.3(a) and 4.3(b). Since S_3^{**} corresponds to a fully failed state of the system, the generation of its corresponding LQN model is unnecessary and hence avoided.

Fig.4.3(a) LQN Model corresponding to S_1^{**}

Fig.4.3(b) LQN Model corresponding to S_2^{**}

Figure 4.3 LQN Models corresponding to system activation states of Fig. 4.1

These LQN models are solved by the LQNS tool [FRAN95, WOOD95b] for determining their performance measures and the *throughput* of the reference task t1 is selected to be the reward to be associated with the corresponding *system activation state* as shown in Table 3.

Table 3: Rewards associated with System Activation States of Table 2

<i>System Activation State</i>	$Prob(S_j^{**})$	Reward $R_i = \text{throughput of task t1}$
S_1^{**}	0.1371	0.6869
S_2^{**}	0.6918	0.9524
S_3^{**}	0.1711	0

Finally, the steady-state performability of the system P_{steady} is calculated as:

$$\begin{aligned}
 P_{steady} &= (0.1371 * 0.6869) + (0.6918 * 0.9524) + (0.1711 * 0) \\
 &= 0.753 / \text{sec.}
 \end{aligned}$$

The next section describes the second strategy which avoids such enumeration of all the *system configuration states*.

4.1.3 Approach 2: State Aggregation Approach

This approach creates all the possible distinct *system activation states* from the corresponding *fault propagation AND-OR graph*, G , and also computes the probability of the system being in each of those states directly without generating/enumerating all the *system configuration states* and also avoids solving the underlying Markov model.

Before describing this approach, some notations and terminologies are provided to be used in the description as follows:

$Child(v)$ denotes the set of children of node v , i.e. $Child(v)$ is the set of nodes w , such that $(v, w) \in E$.

Definition 4.1 A *Status Graph* G_i of the *fault propagation AND-OR graph* $G = (V, E, r, L, priority, \Pi)$ is defined to be a subgraph of G with an additional status value associated with its nodes representing their state (failed or working). Thus $G_i = (V_i', E_i, r, L_i', priority, \Pi)$ where

- $V_i' = \{(v, status(v)) \mid v \in V_i; V_i \subseteq V; status(v) \in \{0, 1\}\}$, $status(v)$ representing the state (0 for failed or 1 for working) of the node v
- $E_i \subseteq E$
- $L_i' \subseteq V_i'$ and $L_i' = \{(l, status(l)) \mid l \in L_i; L_i \subseteq L; status(l) \in \{0, 1\}\}$.

Definition 4.2 A *Status Graph* G_s is termed to be an *Activation Graph*, if the following conditions hold:

- (a) r is the start node of G_s and $\exists v \in Child(r)$ such that $(v, 1) \in V_s'$
- (b) For an *entry* node $v \in V_{entry}$:
 - if $(v, 1) \in V_s'$, then $\forall (w \in Child(v)), (w, 1) \in V_s'$.
 - if instead, $(v, 0) \in V_s'$, then not all $w \in Child(v)$ has $status(w)$ equal to 1.
- (c) For a *service* node $v \in V_{service}$:
 - if $(v, 1) \in V_s'$, then $\exists w \in Child(v)$ with $priority((v, w)) = m$, such that $(w, 1) \in V_s'$ and $\forall (t \in Child(v))$ with $priority((v, t)) < m$, $(t, 0) \in V_s'$.

- if instead, $(v, 0) \in V_s'$, then none of children w of v has $status(w)$ equal to 1.

Each *activation graph* G_s of G corresponds to a distinct *system activation state*, which is found by associating the status values of the *entry* nodes of the *activation graph* to the corresponding entries in the *system activation state* and associating *don't care* values to all other entries which are not in the *activation graph*. Therefore all the possible distinct *system activation states* can be determined by generating all the possible *activation graphs* of G .

This approach is divided into two phases:

I. *Status Graphs Generation Phase*

II. *Activation Graphs Determination and Probability Computation Phase*

Phase - I: Status Graphs Generation Phase

This phase generates a set of *status graphs*, $Z' \supseteq Z$, where Z is the set of all possible *activation graphs* of G , by traversing G once. It involves the following steps:

(1) Traverse G in the breadth-first-search fashion[NILS80]. During traversal, process a node v as follows:

If $v = r$,

For each of the combination of status values 1 and 0 for the children of r ,

Generate a new *status graph* G_i with:

- V_i' consisting of the node r with status 1 (or 0 for all zero combination of status values of its children) and the i -th combination of the children of r with their status values as 1 or 0 and
- E_i being the set of edges connecting r to the children of r .

If $v \in V_{entry}$,

In each *status graph* G_i where $(v, 1) \in V_i'$,

If $\exists w \in Child(v)$ such that $(w, 0) \in V_i'$ then

Abort construction of G_i and remove it from the set of *status graphs*

Else

Expand G_i in such a way that:

- V_i' becomes $V_i' \cup \{(w, 1) | (w \in \text{Child}(v))\}$ and
- E_i becomes $E_i \cup \bigcup_{w \in \text{Child}(v)} (v, w)$.

If $v \in V_{\text{service}}$,

In each *status graph* G_i where $(v, 1) \in V_i'$,

Replace G_i by 0 or more expanded *status graphs* $G_{i,k}$ depending on the children w_k of v , $k = 1, 2, \dots, m$, where $m = |\text{Child}(v)|$

Consider each child w_k of v in turn, for $k = 1, 2, \dots, m$, where w_k denote the child of v with *priority* value k .

If w_k has already been included in G_i with status 0, or if w_j ($j < k$) has been included in G_i with status 1, do not generate $G_{i,k}$

Else

Generate $G_{i,k}$ in such a way that:

- $V_{i,k}'$ becomes $V_i' \cup \{(w_j, 0) | (j < k)\} \cup \{(w_k, 1)\}$ and
- $E_{i,k}$ becomes $E_i \cup \bigcup_{j=1,2,\dots,k} (v, w_j)$.

If $v \in V_{\text{proc}} \cup V_{\text{task}}$, no action is taken.

Z' is the set of *status graphs* found by these steps. The elements of Z' will henceforth be denoted by G_i .

Phase - II Activation Graphs Determination and Probability Computation Phase

This phase tests the condition of each of the *status graphs*, $G_i \in Z'$, generated in Phase-I, being *feasible* or *infeasible* and then computes the probability of each of the *feasible status graphs*. A *feasible status graph* is an *activation graph*. A *status graph* G_i is *infeasible* if it con-

tains an inconsistency between the status of a node v (where) and the status of its descendants.

Definition 4.3 A status graph G_i is defined to be *infeasible* if any of the following conditions hold:

- (i) For an *entry* node v , $(v, 0) \in V_i'$, $\forall w \in Child(v)$, either $(w, 1) \in V_i'$ or *buildFaultTree*(w) equals -1,
- (ii) For a *service* node v , $(v, 0) \in V_i'$, $\exists w \in Child(v)$ such that $(w, 1) \in V_i'$ or *buildFaultTree*(w) equals -1,

where the procedure *buildFaultTree*(w) is described later in this section.

Otherwise, the status graph G_i is *feasible*. The set of *feasible* status graphs will be Z , the set of *activation graphs*.

Then the probability of each of the *activation graphs*, $G_s \in Z$ can be computed as follows:

$$\begin{aligned}
 Prob(G_s) &= \prod_{(v, status(v)) \in V_s'} Prob(v \text{ being in state } status(v)) \\
 &= \prod_{(l, status(l)) \in L_s'} Prob(l \text{ in } status(l)) \cdot \prod_{(v, 0) \in L_s'} Prob(v \text{ in failed state}) \dots (1)
 \end{aligned}$$

$$= \prod_{(l, 1) \in L_s'} \Pi(l) \cdot \prod_{(l, 0) \in L_s'} (1 - \Pi(l)) \cdot \prod_{(v, 0) \in L_s'} Prob(v \text{ in failed state}) \dots (2)$$

Since in the *activation graph* G_s , the *entry* or *service* nodes with status 1 (i.e. $(v, 1) \notin L_s'$) are expanded, the probability calculation does not involve them explicitly; they depend on their children. However for the failed *entry* or *service* nodes (i.e. $(v, 0) \in L_s'$), the probability calculation must be made separately to give *Prob(v in failed state)* in (2), and this must be done using the information contained in G . A fault tree can then be constructed using both G and G_s to model the simultaneous failure of all these nodes $(v, 0) \in L_s'$.

The Phase-II can be realized as follows:

1. Create a dummy node named *fiRoot* such that the children of *fiRoot* are the nodes $(v, 0) \in L_i'$.

2. For each of the child v of the $ftRoot$, call $buildFaultTree(v)$ based on G . If $\exists v \in Child(ftRoot)$ such that $buildFaultTree(v)$ returns -1, then the *status graph* G_i is *infeasible* and no fault tree is created for it. Otherwise include $ftRoot$ as the root in the fault tree represented by an *and* gate whose inputs are the outputs of the gates representing those children v for which $buildFaultTree(v)$ equals 1.

buildFaultTree(v)

Begin

Process node v as follows:

/ v equal to r is an impossible case since there is no activation graph corresponding to this case*/*

Case 1: $v \in V_{entry}$

If $\forall (w \in Child(v))$, either $(w, 1) \in V_i'$ or $buildFaultTree(w)$ equals -1 then

return -1; */* v cannot fail in this case */*

If $\exists w \in Child(v)$ such that $(w, 0) \in V_i'$ then return 0;

If $\exists w \in Child(v)$ where $(w, 1) \notin V_i'$, such that $buildFaultTree(w)$ equals 0 then

return 0;

/ in the above two cases the failure of v is certain due to the failure of w */*

Else

Include v (if not included) in the fault tree as an *or* gate whose inputs are the outputs of the gates representing those children w of v where $(w, 1) \notin V_i'$ and $buildFaultTree(w)$ equals 1;

return 1;

Case 2: $v \in V_{service}$

If $\exists w \in Child(v)$ such that $(w, 1) \in V_i'$ or $buildFaultTree(w)$ equals -1, then

return -1; */* v cannot fail in this case */*

If $\forall (w \in \text{Child}(v)), (w, 0) \in V_i'$ or $\text{buildFaultTree}(w)$ equals 0 then

return 0;

/* in the above case the failure of v is certain due to the failure of all its children */

Else

Include v (if not included) in the fault tree as an *and* gate whose inputs

are the outputs of the gates representing all its children

return 1;

Case 3: $v \in V_{proc} \cup V_{task}$

If $(v, 0) \in V_i'$ then return 0;

Else

Include v (if not included) as a *basic condition* in the fault tree

return 1;

End

An *and* gate or an *or* gate is considered to have at least two inputs. During the construction of the fault tree, if there is only one valid input to an *and(or)* gate, then the other input to the gate is implicitly taken to be logic 1(logic 0).

The fault tree thus constructed can be solved to compute the third product in (2) utilizing the Sum-of-Disjoint-Products algorithm [RAI95, VEER91, ABRA79, HARI87].

In the FTLQNS tool, the fault tree calculation for the third product in (2) has been accomplished by using the SHARPE tool [SAHN96, SAHN87a, SAHN87b]. The tool carries this out by generating the fault-tree description file in SHARPE syntax and then invoking the SHARPE tool on that file.

The probability of each of the *activation graphs* thus computed can then be associated with its corresponding *system activation state*.

The following example illustrates the use of this *State Aggregation Approach*.

4.1.4 Example

To see how well the state aggregation strategy works, let us consider a model of a fault-tolerant client-server system with one reference task as shown in Figure 4.4. In this figure, there are five tasks $t1$ - $t5$ with entries $e1$ - $e5$ running on the processors $p1$ - $p5$. The independent failure probabilities for ($p1$ - $p5$, $t1$ - $t5$) are in order (0.03, 0.05, 0.07, 0.09, 0.11, 0.1, 0.08, 0.06, 0.04, 0.02).

Figure 4.4 An FTLQN Model to illustrate State Aggregation Approach

The FTLQN model in Figure 4.4 is transformed into its corresponding *fault propagation AND-OR graph* as shown in Figure 4.5. *serv1*, *serv2* and *serv3* are the *service* nodes in the *fault propagation AND-OR graph* corresponding to the solid black rectangles on entries $e1$, $e2$ and $e3$ respectively in Figure 4.4.

Figure 4.5 Fault propagation AND-OR graph corresponding to Fig. 4.4

Next, all the possible distinct *system activation states* and the probabilities of the system being in those states are determined from the *fault propagation AND-OR graph* in Figure 4.5 by using the *State Aggregation Approach* as follows:

In Phase-I of the approach, five *status graphs* G_1 , G_2 , G_3 , G_4 and G_5 are generated as shown in Figure 4.6(a), 4.6(b), 4.6(c), 4.6(d) and 4.6(e). In Figure 4.6(a)–4.6(e), a node v is labelled with $(v, \text{status}(v))$. Although a *status graph* is a subgraph of the *fault propagation AND-OR graph* G , and thus inherits the *priority function*, however, the *priority function* is not

necessary to consider in case of a *status graph* since it represents a particular configuration of the system based on the priority values.

Figure 4.6(a) Status Graph G_1

Figure 4.6(b) Status Graph G_2

Figure 4.6 Status graphs of Fig. 4.5

Figure 4.6(c) *Status Graph G_3*

Figure 4.6(d) *Status Graph G_4*

Figure 4.6 Status graphs of Fig. 4.5 (contd.)

Figure 4.6(e) Status Graph G_5

Figure 4.6 Status graphs of Fig. 4.5 (contd.)

In Phase-II, each of the *status graphs* G_1 , G_2 , G_3 , G_4 and G_5 are found to be *feasible*. Thus, all the *status graphs* G_1 , G_2 , G_3 , G_4 and G_5 are the *activation graphs*.

Next, the probability of each of the *activation graphs* G_j , $j = 1, 2, 3, 4, 5$ is calculated as follows:

$Prob(G_1) = \Pi(p_1) \cdot \Pi(t_1) \cdot \Pi(p_2) \cdot \Pi(t_2) \cdot \Pi(p_4) \cdot \Pi(t_4) = 0.6666$. There is no fault tree corresponding to G_1 since G_1 does not contain any node with status 0.

For probability computation of G_2 , G_3 , G_4 and G_5 , the fault trees F_2 , F_3 , F_4 and F_5 respectively are constructed as shown in Figure 4.7(a), 4.7(b), 4.7(c) and 4.7(d).

Figure 4.7 Fault trees corresponding to the activation graphs of Fig. 4.5

Figure 4.7(d) Fault tree F_5 for failure of $e1$, in G_5

Figure 4.7 Fault trees corresponding to the activation graphs of Fig. 4.5 (contd.)

Then, the probabilities of G_2 , G_3 , G_4 and G_5 are computed as:

$$\begin{aligned}
 Prob(G_2) &= \Pi(p1) \cdot \Pi(t1) \cdot \Pi(p3) \cdot \Pi(t3) \cdot \Pi(p4) \cdot \Pi(t4) \cdot (\text{SHARPE's output for } F_2) \\
 &= 0.084
 \end{aligned}$$

$$Prob(G_3) = \Pi(p1) \cdot \Pi(t1) \cdot \Pi(p2) \cdot \Pi(t2) \cdot \Pi(p5) \cdot \Pi(t5) \cdot (\text{SHARPE's output for } F_3) \\ = 0.0841$$

$$Prob(G_4) = \Pi(p1) \cdot \Pi(t1) \cdot \Pi(p3) \cdot \Pi(t3) \cdot \Pi(p5) \cdot \Pi(t5) \cdot (\text{SHARPE's output for } F_4) \\ = 0.0106$$

$$Prob(G_5) = \text{SHARPE's output for } F_5 = 0.1548.$$

Next, the *system activation states* S_1^{**} , S_2^{**} , S_3^{**} , S_4^{**} and S_5^{**} are determined corresponding to the *activation graphs* G_1 , G_2 , G_3 , G_4 and G_5 respectively. The probability of each of the *activation graphs* are then associated with its corresponding *system activation state*.

The *system activation states* and their probabilities thus obtained are shown in Table 4 with the indication of the number of *system configuration states* contained in each of the *system activation states*.

Table 4: System Activation States and their probabilities for Figure 4.4

<i>System Activation State S_i^{**}</i>	$S_i^{**} = \{ [s_1^*, s_2^*, s_3^*, s_4^*, s_5^*] \}$. $s_1^*-s_5^*$ are the <i>operational states</i> of the entries e1-e5 respectively. <i>d</i> represents <i>don't-care</i> .	No. of System Configuration States in S_i^{**}	$Prob(S_i^{**})$
S_1^{**}	$\{ [1, 1, d, 1, d] \}$	16	0.6666
S_2^{**}	$\{ [1, 0, 1, 1, d] \}$	12	0.084
S_3^{**}	$\{ [1, 1, d, 0, 1] \}$	12	0.0841
S_4^{**}	$\{ [1, 0, 1, 0, 1] \}$	9	0.0106
S_5^{**}	System Failed State	$1024 - 49 = 975$	0.1548

An LQN model for each of the *system activation states* S_1^{**} , S_2^{**} , S_3^{**} and S_4^{**} is generated as shown in Figure 4.8(a) - 4.8(d) and all of those models are solved by the LQNS tool [FRAN95, WOOD95b] for determining their performance measures. Since S_5^{**} corresponds to the fully failed state of the system, the generation of its corresponding LQN model is avoided.

Fig4.8(a) LQN Model corresponding to S_1^{**}

Fig4.8(b) LQN Model corresponding to S_2^{**}

Fig4.8(c) LQN Model corresponding to S_3^{**}

Fig4.8(d) LQN Model corresponding to S_4^{**}

Figure 4.8 LQN Models corresponding to system activation states of Fig. 4.4

In this example, the *throughput* of the reference task t1 is selected to be the reward to be associated with the corresponding *system activation state* as shown in Table 5.

Table 5: Rewards associated with System Activation States of Table 4

<i>System Activation State S_i^{**}</i>	<i>Prob(S_i^{**})</i>	<i>Reward $R_i =$ throughput of task t1</i>
S_1^{**}	0.6666	0.3912
S_2^{**}	0.084	0.3912
S_3^{**}	0.0841	0.3912
S_4^{**}	0.0106	0.3912
S_5^{**}	0.1548	0

It is observed from Table 5 that the performance result (reward) is the same for all the *system activation states* S_1^{**} - S_4^{**} . This is because the LQN performance models corresponding to these four states have equal model parameters and semantics. It can therefore be concluded that, the *State Aggregation Approach* could have been further optimized if the equality of the model parameters and semantics among the LQN models could have been detected.

Finally, the expected steady state reward rate, P_{steady} , of the system is obtained as:

$$P_{steady} = 0.331 \text{ /sec.}$$

If the system corresponding to Figure 4.4 is considered to be fully reliable but with no stand-by servers, it becomes similar to the model shown in Figure 4.8(a). In that case, the mean throughput reward of the system is obtained as 0.3912/sec. This is an upper bound for the mean throughput reward of the system in Figure 4.4. Next, if the system corresponding to Fig. 4.4 is considered without the stand-by servers but with the failures, then it becomes similar to the model shown in Fig. 4.8(a) with failures. In this case, the mean throughput reward of the system is obtained as 0.261. It is observed that the mean throughput reward increases from 0.261 to 0.331 by incorporating the stand-by servers in the system having failures as in Figure 4.4.

When this example is worked out by using the *Enumeration Approach*, it enumerates 1024 *system configuration states* and the CPU time taken is 53.7 secs to determine the *system acti-*

vation states and their probabilities whereas the CPU time is 2.8 secs to do the same for the *State Aggregation Approach* by using the FTLQNS tool on a SUN Sparcstation-20. This shows that the *State Aggregation Approach* is more efficient than the *Enumeration Approach*.

The following section discusses the efficiency of the *State Aggregation Approach* over the *Enumeration Approach* for large systems with respect to the CPU time using the FTLQNS tool.

4.2 Comparison of the Approaches: Enumeration and State Aggregation

This section attempts to compare the effort of solving an FTLQN model using the two approaches of the FTLQN performability algorithm by introducing a sequence of stand-by tasks at different parts in a simple client-server system. This is done in order to demonstrate the efficiency of the *State Aggregation Approach* over the *Enumeration Approach*, and the scalability of the former.

Figure 4.9 FTLQN Model of a simple client-server system

Figure 4.10 FTLQN Model with alternative task t4 added to Fig. 4.9 (Stage-I)

Figure 4.11 FTLQN Model with alternative tasks t_5 and t_6 added to Fig.4.10
(Stage-II)

Figure 4.12 FTLQN Model with tasks t_7 , t_8 and t_9 added to Fig. 4.11
(Stage-III)

Let us consider a client-server system as shown in Figure 4.9 which consists of three tasks t_1 , t_2 , t_3 running on their own private processors (not explicitly shown in the figure). The system is enhanced by adding stand-by servers in three stages. In stage-I, a single alternative task t_4 and its own processor, where t_4 acts as the stand-by server for the task t_3 , is added as shown in Figure 4.10. In stage-II, two more alternative tasks t_5 , t_6 and their own private processors are added as shown in Figure 4.11. In stage-III, the system is further enhanced by adding three more tasks t_7 , t_8 and t_9 and their own private processors as shown in Figure 4.12. For simplicity, the independent failure probabilities for all the tasks and the processors in these models have been taken to be 0.1 except the reference task t_1 and its processor which have been assumed to be fully reliable.

Next, the performance of the *Enumeration Approach* and the *State Aggregation Approach* of the FTLQN Solution Algorithm is compared in Table 6 by running the four models. The model solution time (execution time of *State Aggregation Approach* plus the execution times of the LQNS on all the LQN models corresponding to the relevant figure) and the mean throughput reward are also provided for these models. These results were computed using the FTLQNS tool in which both the approaches are implemented. The execution times shown in the table are in seconds (unless and otherwise stated explicitly), measured on a SUN Sparcstation-20. It is observed that there is a marked increase in the CPU time for the *Enumeration Approach* since it has to enumerate a large number of *system configuration states* for a large system. Since the *State Aggregation Approach* avoids such enumeration, the CPU times are much smaller for it. It must also be noted that if one were to formulate a Markov Reward model for Figure 4.12, whose states are the *system configuration states*, it would have 262144 states similar to the *Enumeration Approach*, and it would take a long time to solve.

Table 6: Execution times for System Activation States construction and their probability computation by the *Enumeration Approach* and the *State Aggregation Approach* for the Figures 4.9–4.12

FTLQN Models	No. of System Activation States	No. of System Configuration States	CPU time (in secs) for Enumeration Approach	CPU time (in secs) for State Aggregation Approach (SAA)	FTLQNS Model Solution time (in secs) including SAA	Mean Throughput Reward (/sec)
Fig 4.9	2	64	0.5	0.4	0.6	0.183
Fig 4.10 (Stage-I)	3	256	3.6	1.2	1.6	0.217
Fig 4.11 (Stage-II)	5	4096	902.2	2.9	3.5	0.264
Fig 4.12 (Stage-III)	10	262144	~ 66 hours (only enumerated 15% of states in 11 hrs)	6.8	8.7	0.276
Figure 4.9 with no failures (perfectly reliable service)						0.278

If the system in Figure 4.9 is considered to be a fully reliable system, then its mean throughput (reward) is obtained to be 0.278 as shown at the bottom of Table 6. This is an upper bound for the mean throughput reward of the systems with failures. It is observed from Table 6 that as the number of stand-by servers are increased, the mean throughput reward increases to 0.276 for Figure 4.12 which is close to 0.278, the mean throughput reward of Figure 4.9 with no failures.

The following section describes the tool that automates the FTLQN performability modeling technique.

4.3 Fault-Tolerant Layered Queueing Network Solver (FTLQNS)

This section presents a high level description of the FTLQNS tool that automates the FTLQN performability modelling technique. This tool implements both the *State Aggregation Approach* and the *Enumeration Approach*.

4.3.1 Architecture of FTLQNS

Figure 4.13 shows the high level architecture of the tool. The arrowheads indicates the direction of control/data flow. The FTLQNS Parser takes an FTLQN model description that obeys the BNF given in Appendix A and develops the corresponding *fault-propagation AND-OR graph*. The AND-OR Graph Analyzer (a C++ Engine) operates on the *fault-propagation AND-OR graph* and produces the model input files for the LQNS [FRAN95, WOOD95b] corresponding to the distinct *system activation states* and computes the probabilities of the system being in those states. In the case of the *State Aggregation Approach*, for the computation of the probabilities, the AND-OR Graph Analyzer generates the fault trees for the *system activation states* in order to find the third product in equation (2) of Section 4.1.3. The SHARPE tool is then iteratively invoked for the fault tree calculations. However, in the case of *Enumeration Approach*, the AND-OR Graph Analyzer generates and enumerates all the *system configuration states* to determine the distinct *system activation states* and their probabilities.

Next the LQNS tool is iteratively invoked for finding the performance metric to be used as a reward corresponding to each of the *system activation states*. Finally the probabilities of the *system activation states* obtained from the AND-OR Graph Analyzer and the results obtained from the LQNS tool are fed to the Performability Calculator to compute the steady-state performability.

The database is a collection of UNIX files which stores the data required for input to the FTLQNS parser, input to the SHARPE tool, input to the LQNS tool, output of the LQNS tool, output generated by the SHARPE tool, input to the Performability Calculator and the output generated by it.

Figure 4.13 Architecture of the FTLQNS tool

The next subsection describes the usage of the FTLQNS tool.

4.3.2 How to use FTLQNS

Figure 4.14 presents the flowchart of the FTLQNS tool. The lightly shaded rounded-rectangles represent the data. The deeply shaded rectangles represent the implementations carried out in this thesis. The unshaded rectangles represent the existing tools. The sequence of steps of how the steady-state performability of a system is obtained from its model description in FTLQNS syntax (given in Appendix A) is represented by the numbers on the arrows in the flowchart. The steps 4-7 are not carried out in case of *Enumeration Approach*. They are carried out for the *State Aggregation Approach* only.

Figure 4.14 Flowchart of FTLQNS tool

The FTLQNS tool is invoked as follows:

```
% ftlqns [-s] [-e] filename
```

The *filename* should have *ftlqn* extension, for example, *foo.ftlqn*. The description in the file follows the BNF format given in Appendix A. The option -s stands for invoking the *State Aggregation Approach* and the option -e stands for invoking the *Enumeration Approach*. The default option is -s.

Next, a sample usage of the FTLQNS tool is demonstrated using the example given in the sub-section 4.1.4. The input file *ex414.ftlqn* corresponding to the FTLQN model in Figure 4.4 is shown in Figure 4.15.

```
G "ex414.ftlqn" 1e-06 50 5 0.90 -1

P 5
p p1 f (0.03)
p p2 f (0.05)
p p3 f (0.07)
p p4 f (0.09)
p p5 f (0.11)
-1

T 5
t t1 r e1 -1 p1 (0.1)
t t2 n e2 -1 p2 (0.08)
t t3 n e3 -1 p3 (0.06)
t t4 n e4 -1 p4 (0.04)
t t5 n e5 -1 p5 (0.02)
-1

E 5
s e1 0.0 1.0 -1
s e2 1.0 1.0 -1
s e3 1.0 1.0 -1
s e4 1.0 1.0 -1
s e5 1.0 1.0 -1
ALT y e1 [e2, e3] 0.6 0.0 -1
ALT y e2 [e4, e5] 0.8 0.0 -1
ALT y e3 [e4, e5] 0.8 0.0 -1
-1
```

Figure 4.15 Input file *ex414.ftlqn* corresponding to FTLQN model in Fig. 4.4

The FTLQNS is invoked on the file ex414.flqn as follows:

```
% flqns -s ex414.flqn
```

On running flqns using the *State Aggregation Approach* as shown above, the sample intermediate output data obtained corresponding to B, C, D, E and F of the flowchart in Figure 4.14 is provided in Appendix B. The final output generated is shown in Figure 4.16 below.

Steady-State Performability of the system: 0.331

Figure 4.16 Output generated by FTLQNS for Figure 4.15

So far, the FTLQN model solution strategies for the computation of its steady-state performability have been presented and the solver that implements the strategies have been discussed. The next section describes the computation of the point performability of an FTLQN model. The computation of point performability has not been accommodated in the FTLQNS tool.

4.4 Computation of Point Performability

The point performability, $P_{instant}(t)$, can be obtained as:

$$P_{instant}(t) = \sum_{j=1}^n R_j Prob_t(S_j^{**}), \text{ where } Prob_t(S_j^{**}) \text{ represents } Prob(S_j^{**}) \text{ at time } t,$$

from an FTLQN model by the FTLQN performability algorithm described in Section 4.1 of this chapter where in Step (2), instead of computing the steady-state probabilities $Prob(S_j^{**})$, the transient probabilities $Prob_t(S_j^{**})$ need to be computed.

In the model, each entity i is governed by a separate Markov chain with a working state ($s_i = 1$) and a failed state ($s_i = 0$), with failure rate λ and repair rate μ which gives its transient probabilities (using the initial condition that the entity i is in state 1 at time 0) $Prob(s_i(t) = 0)$ and $Prob(s_i(t) = 1)$ as:

$$Prob(s_i(t) = 0) = \frac{\lambda}{\mu + \lambda} + \frac{\lambda(e^{-(\mu + \lambda)t})}{\mu + \lambda}$$

$$Prob(s_i(t) = 1) = \frac{\mu}{\mu + \lambda} + \frac{\lambda(e^{-(\mu + \lambda)t})}{\mu + \lambda}$$

Thus for this case, the function Π in the *fault propagation AND-OR graph* G also becomes a function of t as $\Pi(l, t)$ which represents the probability of node l being in working state at time t . Since the failure of hardware and software components are assumed to be independent, therefore, the computation of $Prob_i(S_j^{**})$ can be achieved by the product formula in equation (2) given in the *State Aggregation Approach* described in Section 4.1.3 where Π becomes a function of time t as well.

If the entities in the system cannot be repaired on failure, their steady-state probability of being operational becomes zero. In that case, the measure of point performability becomes significant instead of the steady-state measure.

4.5 Summary

This chapter has introduced two new solution strategies, *Enumeration* and *State Aggregation Approaches*, for the FTLQN model in order to compute its steady-state performability. It has been demonstrated with examples that the *State Aggregation Approach* is more efficient than the *Enumeration Approach*. It has also presented the high-level description of the FTLQNS tool that automates the FTLQN performability modelling technique and implements both the solution strategies. Besides, it has provided a brief description about the computation of the point performability for an FTLQN model.

Chapter 5 Application: A Call Processing System

In this chapter, the application of the *State Aggregation Approach* of the FTLQN Performance Algorithm on a large prototype call processing system studied by [JOGL97] is presented.

5.1 Call Processing System

The call processing system, as described in [JOGL97], uses a “two half-call” model [MCDO90] for call processing in which the system is partitioned into two logical parts, one corresponding to the call originating site and the other corresponding to the call terminating site. The basic functions performed by the call-processing software are as follows:

- detecting a request for service
- interpreting the dialed digits
- alerting the called party
- establishing a connection
- disconnecting a call
- charging

The call processing system is implemented using a message oriented middleware based on a fast ATM network Figure 5.1 [JOGL97] shows the hardware architecture of the prototype call processing system. The hosts (Power PCs) are connected to an ATM network. Each host has the call processing software that supports the two half-call model.

Figure 5.1 Call Processing System's Hardware Infrastructure

The next section describes the Layered Queueing Network Model of this call processing system.

5.2 Layered Queueing Network Model of the System

A Layered Queueing Network Model of the call processing system was described in [JOGL97], as shown in Figure 5.2. The model in Figure 5.2 represents the tasks involved in call processing at each of the call processing sites in the system. In the model, the software objects modelling the components on the originating half are the call originating tasks and those on the terminating half are the call terminating tasks.

The ATM network and the Integrated Device Driver that enables the application to directly send messages on the ATM network are modelled as delay servers. All the process in the system are assumed to have infinite number of threads. There are 182638 clients in the system each residing on a separate CPU. All the tasks and the processors in this model are fully reliable. For details about the structure of the LQN model refer to [JOGL97].

The next section provides the FTLQN model of the system.

Figure 5.2 Layered Queueing Network model for Call Processing System

5.3 Fault-Tolerant Layered Queueing Network Model of the System

Figure 5.3 illustrates the FTLQN model of the call processing system with no standbys. It is the extension of the LQN model in Figure 5.2, with failures indicated. The failure probabilities of the tasks and the processors which are other than zero are written above them. In this model, it is assumed that the tasks Database_Manager, Location_Server, Orig_Agent_Factory, Term_Agent_Factory, ATM_NET and the processors on which they run can undergo failure. The failure probability of the Database_Manager has been taken to be the highest because of its sheer complexity.

The tasks Orig_Terminal_Agent, Orig_User_Agent and Orig_Call_Setup are instantiated by the Orig_Agent_Factory and hence are assumed to be fully reliable since they can be re-instantiated on failure by the Orig_Agent_Factory. A similar reason holds for the assumption that the tasks Term_Terminal_Agent, Term_User_Agent and Term_Call_Setup are fully reliable. The task Trans_Overhead modeling the transmission overhead in hardware or a network adapter is assumed to be fully reliable. Besides, the tasks Orig_Integrated_Device_Driver and Term_Integrated_Device_Driver are also assumed to be fully reliable.

The following section demonstrates the application of the *State Aggregation Approach* of the FTLQN Performability algorithm to the call processing system in Figure 5.3 when standby components are introduced.

5.4 Evaluation of the State Aggregation Approach

The FTLQN model in Figure 5.3 is enhanced in five stages by adding standby servers for those tasks which can undergo failure. Each stage will be evaluated, to see what its features contribute.

In stage-I, a single alternative task, Database_Manager_2 and its own processor, CPU2A is added to the model in Figure 5.3 where Database_Manager_2 acts as the standby server for the task Database_Manager as shown in Figure 5.4. The rest of the configuration (and the failure probabilities) remains the same as in Figure 5.3.

In stage-II, another alternative task, Location_Server_2 and its own processor CPU4A is added where Location_Server_2 acts as the stand-by server for the task Location_Server as shown in Figure 5.5. The rest of the configuration (and the failure probabilities) remains the same as in Figure 5.4.

In stage-III, the system is enhanced by adding one alternative task, Term_Agent_Factory_2 (stand-by for Term_Agent_Factory) and three other tasks Term_Terminal_Agent_2 (TTA2), Term_User_Agent_2 (TUA2) and Term_Call_Setup_2 (TCS2) as shown in Figure 5.6. All of these tasks run on the processor CPU3A. The reason for adding the three tasks, TTA2, TUA2 and TCS2 is as follows: If the task Term_Agent_Factory fails, then the tasks Term_Terminal_Agent, Term_User_Agent and Term_Call_Setup which were instantiated by Term_Agent_Factory, fail as well. In this figure, the rest of the configuration (and the failure probabilities) remains the same as in Figure 5.5.

In stage-IV, the system is further enhanced by adding one alternative task, Orig_Agent_Factory_2 (stand-by for Orig_Agent_Factory) and three other tasks Orig_Terminal_Agent_2 (OTA2), Orig_User_Agent_2 (OUA2) and Orig_Call_Setup_2 (OCS2) as shown in Figure 5.7. All of these tasks run on the processor CPU1A. The reason for adding the three tasks, OTA2, OUA2 and OCS2 is similar to the reason provided above for the addition of TTA2, TUA2 and TCS2 in stage-III. In this figure, the rest of the configuration

(and the failure probabilities) remains the same as in Figure 5.6.

In stage-V, a stand-by server, ATM_NET_2, for the network task ATM_NET is added as shown in Figure 5.8. The task ATM_NET_2 runs on the processor NETCPUA. The rest of the configuration (and the failure probabilities) remains the same as in Figure 5.7.

Figure 5.4 FTLQN Model for Call processing System (CPS) with stand-by for Database_Manager (Stage-I)

Figure 5.5 FTLQN Model for CPS with stand-bys for Database_Manager and Location_Server (Stage-II)

Figure 5.6 FTLQN Model for CPS with stand-bys for Database_Manager, Location_Server and Term_Agent_Factory (Stage-III)

Figure 5.7 FTLQN Model for CPS with stand-bys for Database_Manager, Location_Server, Term_Agent_Factory and Orig_Agent_Factory (Stage-IV)

Figure 5.8 FTLQN Model for CPS with stand-bys for Database_Manager, Location_Server, Term_Agent_Factory, Orig_Agent_Factory and ATM_NET (Stage-V)

Table 7: Execution times for System Activation States construction, their probability computation and the Model Solution by using the *State Aggregation Approach* for the Figures 5.3-5.8

FTLQN Models	No. of System Activation States	No. of System Configuration States	CPU time (in secs) for <i>State Aggregation Approach</i> (SAA)	FTLQNS Model Solution time (in secs) including SAA	Mean Throughput Reward (/sec)
Fig 5.3	2	16777216	0.4	4.3	0.240
Fig 5.4 (Stage-I)	3	67108864	6.5	14.1	0.266
Fig 5.5 (Stage-II)	5	2.68435×10^8	26.0	41.4	0.282
Fig 5.6 (Stage-III)	9	8.58993×10^9	53.3	83.7	0.288
Fig. 5.7 (Stage-IV)	17	2.7488×10^{11}	118.2	179.9	0.294
Fig. 5.8 (Stage-V)	33	1.0995×10^{12}	319.6	447.6	0.299
Figure 5.3 with no failures (perfectly reliable service)					0.304

The performance of the *State Aggregation Approach* is evaluated in Table 7 by running the six models illustrated in Figures 5.3-5.8. The model solution time (execution time of *State Aggregation Approach* plus the execution times of the LQNS on all the LQN models corresponding to the relevant figure) and the mean throughput reward are also provided for these models. These results were computed using the FTLQNS tool. The execution times shown in the table are in seconds, measured on a SUN Sparcstation-20. Since the *State Aggregation Approach* finds the *system activation states* (i.e. the equivalence classes of *system configuration states*) directly without enumeration, the CPU times for this approach are significantly small.

From Table 7, it is observed that by incorporating stand-by servers for the tasks which can fail, the mean throughput reward of the system increases. It is found that the increase in the

mean throughput reward is the highest ($= 0.266 - 0.240$) when the stand-by task for the Database_Manager is added to the system. This is because the failure probability of the Database_Manager is the highest.

If the system in Figure 5.3 were a fully reliable system, then its mean throughput (reward) would be 0.304 as shown at the bottom of Table 7. This is an upper bound for the mean throughput reward of the system with failures. It is observed from Table 7 that for Figure 5.8 which has the maximum number of stand-by servers, the mean throughput reward is 0.299 which is very close to 0.304, the mean throughput reward of Figure 5.3 with no failures.

From this section, it can be concluded that the *State Aggregation Approach* works efficiently for large systems. It is observed from Table 7 that as the system becomes larger from stage-I to stage-V, the number of *system activation states* and correspondingly the CPU time for the *State Aggregation Approach* increases but does not explode. For the largest system considered in stage-V (Figure 5.8), the CPU time is approximately 320 seconds, which is reasonable.

5.5 Summary

In this chapter, the application of the FTLQN Performability Algorithm on a prototype call processing system has been presented. It has been demonstrated that the *State Aggregation Approach* works efficiently for large systems. It is also observed that as the stand-by servers are introduced in the system with failures, the mean throughput reward increases.

Chapter 6 Extension of FTLQN Model for Modelling Load-Balancing Degradable Systems

A system may be made fault-tolerant by incorporating redundant servers for a service request which may all be used at the same time, with the load being equally divided in order to increase the system processing power. There is a mechanism to balance the load among the remaining servers in case of a failure. This concept may be termed as “load-balancing redundancy concept” and the systems with the above property may be called “load-balancing degradable system”.

This chapter presents the extension to the FTLQN model definition necessary to model a fault-tolerant client-server system involving load-balancing redundancy concept. The failure of the hardware and the software tasks are assumed to be independent. It also presents the solution algorithm for the extended FTLQN model (termed as “load-balanced” FTLQN model) in order to compute its steady-state performability. It further provides an example illustrating the use of the algorithm. The algorithm presented in this chapter, however, has not been implemented and evaluated.

6.1 Extension of FTLQN Model Definition

The FTLQN model as described in [WOOD96] is a performability model of client-server system involving the stand-by redundancy concept. In order to model a fault-tolerant client-server system involving the load balancing redundancy concept, the FTLQN model needs to be changed. This section describes the extended FTLQN model termed as “load-balanced” FTLQN model.

In a load-balanced FTLQN model, a service request arc may be replaced by a set of arcs which can all be used at the same time with the load being equally distributed among them. Figure 6.1 shows a load-balanced FTLQN model for a load-balancing degradable system. Such a set of arcs is shown attached to a rectangle with thick black border on Entry 2, with the

service load written beside it. That is, Entry 2 gets this service from both Entry 4a and Entry 4b with the load being equally divided among them, but if one of them fails, it will switch to the other with the total service load.

Figure 6.1 A Load-balanced FTLQN Model for a load-balancing degradable system

The graph representation of a load-balanced FTLQN model would be the same as described in Section 3.3 of Chapter 3 except that there would be no *priority* function associated with its corresponding *fault propagation AND-OR graph*, G . This means G would now become a five tuple, $G = (V, E, r, L, \Pi)$ where V, E, r, L and Π have the same definition as described in Section 3.3 except that the definition of *service* node now becomes as follows:

- “a service node is an OR node that describes the redundant targets (having no preference order associated with them) for a requested service. It corresponds to a rectangle with thick black border in the load-balanced FTLQN model and is shown with a thick black border in G ”.

This G is henceforth termed as *load-balanced fault propagation AND-OR graph*.

Figure 6.2 shows the *load-balanced fault propagation AND-OR graph* corresponding to the load-balanced FTLQN model in Figure 6.1.

Figure 6.2 Load-Balanced Fault Propagation AND-OR graph corresponding to Fig. 6.1

The following section describes the solution algorithm for a load-balanced FTLQN model in order to compute its steady-state performability measure.

6.2 Performability Algorithm for Load-Balanced FTLQN Model

The performability algorithm for the load-balanced FTLQN model would be the same as described in Section 4.1 of Chapter 4 except that the approaches for achieving step 2 needs to be modified. Since the *State Aggregation Approach* is more efficient than the *Enumeration Approach*, this section presents only the modification of the *State Aggregation Approach*, which is here termed as *load-balanced State Aggregation Approach*, for achieving step 2. The modification is required in Phase-I (i.e. *Status Graphs Generation Phase*) in the processing of a node $v \in V_{service}$ during the graph traversal which is given as follows:

If $v \in V_{service}$,

In each *status graph* G_i where $(v, 1) \in V_i'$,

Replace G_i by 0 or more expanded *status graphs* $G_{i,k}$ depending on the k -th combination of status values 0 and 1 for the children of v (except all zero combination) as follows:

If $\exists w \in Child(v)$ such that w is present in V_i' with status 0(1) although its status value in the k -th combination is 1(0), then do not generate $G_{i,k}$

Else

Generate $G_{i,k}$ in such a way that:

- $V'_{i,k}$ becomes $V_i' \cup \{children\ of\ v\ with\ their\ status\ values\ as\ 1\ or\ 0\ corresponding\ to\ the\ k-th\ combination\}$ and
- $E_{i,k}$ becomes $E_i \cup \bigcup_{w \in Child(v)} (v, w)$.

The next section illustrates the use of the above solution algorithm for a load-balanced FTLQN model by taking an example of a system designed using the load balancing redundancy concept. Since the above algorithm has not been implemented, the illustration and the analysis of the example have been carried out manually. Further, it compares the performability metric obtained by this type of design with the performability metric obtained by designing the same system using the standby-redundancy concept.

6.3 Example

Let us consider a model of a load-balancing degradable client-server system (whose corresponding standby redundancy design has been provided in Figure 4.4) with one reference task t_1 as shown in Figure 6.3. In this figure, there are five tasks t_1 - t_5 with entries e_1 - e_5 running on the processors p_1 - p_5 . Suppose the independent failure probabilities for (p_1 - p_5 , t_1 - t_5) are in order (0.03, 0.05, 0.07, 0.09, 0.11, 0.1, 0.08, 0.06, 0.04, 0.02).

Figure 6.3 A Load-balanced FTLQN Model to illustrate load-balanced State-Aggregation Approach

The load-balanced FTLQN model in Figure 6.3 is transformed into its corresponding *load-balanced fault propagation AND-OR graph* as shown in Figure 6.4. *serv1*, *serv2* and *serv3* are the *service nodes* in the *load-balanced fault propagation AND-OR graph* corresponding to the rectangles with thick black border on entries e_1 , e_2 and e_3 respectively in Figure 6.3.

Figure 6.4 Load-Balanced Fault propagation AND-OR graph corresponding to Fig. 6.3

Next, all the possible distinct *system activation states* and the probabilities of the system being in those states are determined from the *load-balanced fault propagation AND-OR graph* in Figure 6.4 by using the *load balanced State Aggregation Approach* described in Section 6.2 as follows:

In Phase-I of the approach, ten *status graphs* $G_1, G_2, G_3, G_4, G_5, G_6, G_7, G_8, G_9$ and G_{10} are generated as shown in Figure 6.5(a), 6.5(b), 6.5(c), 6.5(d), 6.5(e), 6.5(f), 6.5(g), 6.5(h), 6.5(i) and 6.5(j).

Figure 6.5(a) Status Graph G_1

Figure 6.5(b) Status Graph G_2

Figure 6.5(c) Status Graph G_3

Figure 6.5(d) Status Graph G_4

Figure 6.5 Status graphs of Fig. 6.4

Figure 6.5(e) Status Graph G_5

Figure 6.5(f) Status Graph G_6

Figure 6.5(g) Status Graph G_7

Figure 6.5(h) Status Graph G_8

Figure 6.5 Status graphs of Fig. 6.4 (contd.)

Figure 6.5(i) Status Graph G_9

Figure 6.5(j) Status Graph G_{10}

Figure 6.5 Status graphs of Fig. 6.4 (contd.)

In Figure 6.5(a)-6.5(j), a node v is labelled with $(v, \text{status}(v))$.

In Phase-II, each of the *status graphs* $G_1 - G_{10}$ are found to be *feasible*. Thus, all the *status graphs* $G_1 - G_{10}$ are the *activation graphs*. The probabilities of $G_1 - G_{10}$ are respectively obtained as 0.0107, 0.0106, 0.0731, 0.0107, 0.0106, 0.0733, 0.0745, 0.0735, 0.5082, 0.1548, through the use of the SHARPE tool [SAHN96] and hand calculation.

Next, the *system activation states* $S_1^{**} - S_{10}^{**}$ are determined corresponding to the *activation graphs* $G_1 - G_{10}$ respectively. The probability of each of the *activation graphs* are then associated with its corresponding *system activation state*.

The *system activation states* and their probabilities thus obtained are shown in Table 8.

Table 8: System Activation States and their probabilities for Figure 6.3

<i>System Activation State S_i^{**}</i>	$S_i^{**} = \{ [s_1^*, s_2^*, s_3^*, s_4^*, s_5^*] \}$. s_1^* - s_5^* are the <i>operational states</i> of the entries e1-e5 respectively.	$Prob(S_i^{**})$
S_1^{**}	$\{ [1, 1, 0, 1, 0] \}$	0.0107
S_2^{**}	$\{ [1, 1, 0, 0, 1] \}$	0.0106
S_3^{**}	$\{ [1, 1, 0, 1, 1] \}$	0.0731
S_4^{**}	$\{ [1, 0, 1, 1, 0] \}$	0.0107
S_5^{**}	$\{ [1, 0, 1, 0, 1] \}$	0.0106
S_6^{**}	$\{ [1, 0, 1, 1, 1] \}$	0.0733
S_7^{**}	$\{ [1, 1, 1, 1, 0] \}$	0.0745
S_8^{**}	$\{ [1, 1, 1, 0, 1] \}$	0.0735
S_9^{**}	$\{ [1, 1, 1, 1, 1] \}$	0.5082
S_{10}^{**}	System Failed State	0.1548

In this example the set of arcs emanating from a rectangle with thick black border represent a single service. It implies that when they are used simultaneously, the service load is divided equally among them. Following this implication, an LQN model for each of the *system activation states* S_1^{**} - S_9^{**} is generated as shown in Figure 6.6(a) - 6.6(i) and all of these models are solved by the LQNS tool [FRAN95, WOOD95b] for determining their performance measures. Since S_{10}^{**} corresponds to the fully failed state of the system, the generation of its corresponding LQN model is avoided. It has been observed that the number of *system activation states* in this case is more than that when the standby redundancy design is considered (Table 4). This is because in the case of load-balancing redundancy design, the redundant servers may all be used at the same time as opposed to the standby redundancy design where a backup server is not used until the primary server fails.

Fig6.6(a) LQN model corresponding to S_1^{**}

Fig6.6(b) LQN model corresponding to S_2^{**}

Fig6.6(c) LQN model corresponding to S_3^{**}

Fig6.6(d) LQN model corresponding to S_4^{**}

Figure 6.6 LQN Models corresponding to system activation states of Fig.6.3

Fig6.6(e) LQN model corresponding to S_5^{**}

Fig6.6(f) LQN model corresponding to S_6^{**}

Fig6.6(g) LQN model corresponding to S_7^{**}

Fig6.6(h) LQN model corresponding to S_8^{**}

Figure 6.6 LQN Models corresponding to system activation states of Fig.6.3 (contd.)

Fig6.6(i) LQN model corresponding to S_9^{**}

Figure 6.6 LQN Models corresponding to system activation states of Fig.6.3 (contd.)

In this example, the *throughput* of the reference task t1 is selected to be the reward to be associated with the corresponding *system activation states* as shown in Table 9.

Table 9: Rewards associated with System Activation States of Table 8

System Activation State S_i^{**}	$Prob(S_i^{**})$	Reward $R_i =$ <i>throughput</i> of task t1
S_1^{**}	0.0107	0.3912
S_2^{**}	0.0106	0.3912
S_3^{**}	0.0731	0.4020
S_4^{**}	0.0107	0.3912
S_5^{**}	0.0106	0.3912
S_6^{**}	0.0733	0.4020
S_7^{**}	0.0745	0.4167
S_8^{**}	0.0735	0.4167
S_9^{**}	0.5082	0.4283

Table 9: Rewards associated with System Activation States of Table 8

<i>System Activation State S_i^{**}</i>	<i>Prob(S_i^{**})</i>	<i>Reward $R_i =$ throughput of task t1</i>
S_{10}^{**}	0.1548	0

Finally, the steady-state performability of the system is calculated as:

$$P_{steady} = 0.355/\text{sec}.....(\text{A})$$

If the system, whose extended FTLQN model is given in Figure 6.3, is designed using the standby-redundancy concept, then its corresponding FTLQN model could be illustrated as in Figure 4.4 of the previous chapter. The steady-state performability for this model has been obtained in Section 4.1.4 of Chapter 4 as:

$$P_{steady} = 0.331/\text{sec}.....(\text{B})$$

It is found from the equations (A) and (B) above that, as expected, the performability of the system is lower when the system is designed using the standby-redundancy concept instead of designing it using the load-balancing redundancy concept.

Effect of Multi-threading

Next, the effect of multi-threading in load-balancing scenarios is discussed. When the system in Figure 6.3 is fully operational, the bottleneck is at level 2 (i.e. the level at which the tasks t2 and t3 lies in Figure 6.3) due to the tasks at that level being single-threaded. This is because a task at level 2, in this case, must finish with one response before it starts the next, which effectively sequentializes the execution. By increasing the threading levels at level 2, the system throughput can be increased [NEIL95]. The rule of thumb [NEIL95] suggests that when there is no phase-2 work (as in this case), the number of threads at a level should be the sum of the threads in all the tasks at the next level down plus one to avoid idle time on a task due to blocking. Following this rule of thumb, the throughput of the system in Figure 6.3 can be improved by considering 7 threads of task t1 (at level 1), 3 threads of task t2 and 3 threads of task t3 (at level 2) while retaining the threading level at level 3 to be one. In this multi-threaded case, the probabilities of the *system activation states* as shown in Table 9 remains the same while the *throughput* of the reference task t1 changes as shown in Table 10.

Table 10: Change in the Rewards of Table 9 due to multi-threading

<i>System Activation State S_i^{**}</i>	<i>Prob(S_i^{**})</i>	<i>Reward $R_i =$ throughput of task t1</i>
S_1^{**}	0.0107	0.6677
S_2^{**}	0.0106	0.6677
S_3^{**}	0.0731	0.7116
S_4^{**}	0.0107	0.6677
S_5^{**}	0.0106	0.6677
S_6^{**}	0.0733	0.7116
S_7^{**}	0.0745	0.8141
S_8^{**}	0.0735	0.8141
S_9^{**}	0.5082	0.8340
S_{10}^{**}	0.1548	0

The steady-state performability for the load-balancing scenario due to multi-threading becomes:

$$P_{steady} = 0.677/\text{sec}.....(C)$$

If the same level of multi-threading is carried out on the FTLQN model shown in Figure 4.4 (which is the same system as in Figure 6.3, but designed using standby-redundancy concept), then the steady-state performability is obtained (by using FTLQNS tool) as:

$$P_{steady} = 0.564/\text{sec}.....(D)$$

It is observed that the difference between (D) and (C) is more than the difference between (B) and (A). It can therefore be concluded that the load-balancing redundancy concept is more effective in the multi-threaded case. This is because in the load-balancing redundancy case, there may be configurations such as Figure 6.6(i) where if entry e1 is multi-threaded, then while a thread is blocked for the service from entry e2, another thread may get service from entry e3 resulting in the increase of the system throughput.

6.4 Summary

In this chapter, an extension of the FTLQN model, termed as load-balanced FTLQN model, has been introduced to model the load-balancing degradable systems. A solution algorithm for the extended model has been provided. The solution algorithm has been illustrated by an example. Through the same example, the two designs, one involving the stand-by redundancy concept and the other involving the load-balancing redundancy concept have been compared.

Chapter 7 Conclusion

7.1 Research Summary

An efficient solution approach known as the *State Aggregation Approach*, has been developed, implemented and evaluated for the FTLQN performability model [WOOD96] that involves the stand-by redundancy concept. This approach avoids solving large Markov models. It assumes independent failures of the hardware and the software tasks in the system. This assumption is valid since a crash of a software component may be independent of the failure of its associated hardware. However, the dependencies in the system still exists through the chains of interactions in the layered structure of the FTLQN model.

The *State Aggregation Approach* has been described to compute the steady-state performability of an FTLQN model. It uses the AND-OR graph representation of the FTLQN model to determine the *system activation states* (a *system activation state* being the set of *system configuration states* having the same performance model) directly without enumerating all the *system configuration states*, thus minimizing the number of performance solution required, and to obtain the probabilities of these *system activation states*. For the probability calculation of these states, this algorithm uses the fault tree [SAHN96] based calculations which in turn use a combinatorial sum-of-disjoint product algorithm [RAI95, VEER91, ABRA79, HARI87]. Thus the complexity of the *State Aggregation Approach* is exponential in the size of the embedded fault trees. This approach has been compared with another approach known as *Enumeration Approach* which involves the enumeration of all the *system configuration states* ($= 2^N$ for system with N entities). From the results in Chapter 4, it has been observed that the *State Aggregation Approach* is efficient over the *Enumeration Approach*, although both the approaches have exponential complexity. Although for small systems, the difference between the CPU time of the two approaches is not large, as bigger and bigger systems are considered, the CPU time for the *State Aggregation Approach* has been found to be very pragmatic. For example, for a system with 9 processors and 9 tasks (Figure 4.12) the CPU time for this approach is 6.8 seconds (Table 6) implying the approach to be approximately 34900 times faster than the *Enumeration Approach* in this case.

A large call processing system prototype has been analyzed using the *State Aggregation Approach* and it has been observed that for such a system with $\sim 10^{12}$ *system configuration states*, there are only 33 *system activation states* due to which the *State Aggregation Approach* efficiently worked for such a large system. It can therefore be concluded that for a system with dozens of entities and lots of failures, the time taken by the *State Aggregation Approach* is very realistic and hence for analyzing very large systems, the usage of this approach is highly recommended.

A tool known as Fault Tolerant Layered Queueing Network Solver has been developed to automate the FTLQN technique in which the implementation of both approaches have been embedded.

The FTLQN model has been extended in a way so that it can model the systems involving load-balancing redundancy concept. A solution algorithm has been developed for this model but it has not been implemented. It has been found that if a system is designed using the load-balancing redundancy concept its performability is higher than if it were designed using the stand-by redundancy concept since in the former case the redundant servers can all be used at the same time, contributing to an increase in the number of *system activation states*, with the load being divided equally among them.

7.2 A Possible Generalization of this Work

In this work the dependability modelling of a fault-tolerant system has been carried out using the AND-OR graph analysis in order to avoid the generation and the solution of underlying Markov model. It has been assumed here that the failures of the entities are independent of each other. Each of the entities of the system (software tasks and processors) is governed by a two-state (working or failed) Markov chain with the rates of failure and repair which gives its probabilities of being in working or failed state. Thus, the solution approach presented in this thesis is fundamentally a Markov-Reward based approach. This solution approach can be generalized to a semi-Markov-Reward based approach by making each entity governed by a two-state semi-Markov chain instead of a Markov chain while retaining the assumption of stochastic independence of the entities as before. The working/failure probabilities of the entities in this case can be obtained as follows:

Suppose the sojourn time distributions in states 0 (failed) and 1 (working) be $H_0(t)$ and $H_1(t)$ and the mean sojourn times be $t_i = \int_0^\infty (1 - H_i(t))dt$, $i = 0, 1$, then the steady-state probabilities are:

$$Prob(s_i = 0) = \frac{t_0}{t_0 + t_1} \text{ and } Prob(s_i = 1) = \frac{t_1}{t_0 + t_1}$$

7.3 Future Research

In order to calculate the probability of the *system activation states*, the *State Aggregation Approach* uses the fault tree calculations [SAHN96] which in turn uses a combinatorial sum-of-disjoint product algorithm [RAI95, VEER91, ABRA79, HARI87] making itself exponential in complexity. Further investigation is necessary to optimize these probability calculations.

In the current implementation of the FTLQN solver, only the alternative targets for the services (provided by entries) have been considered. The implementation may be further extended to accommodate alternative target processors for a single task.

The *load-balanced* FTLQN modelling technique has only been described but not automated in the present work. It may therefore be considered to implement this technique and evaluate the *load-balanced State Aggregation Approach* described in Section 6.2.

In this work, the reward rate function has been considered to be uni-dimensional. For the case of multiple classes in the system, the reward rate function may be multi-dimensional. The development of such a multi-dimensional reward rate function may be considered in future investigations.

References

- [ABRA79] J. A. Abraham, "An Improved Algorithm for Network Reliability", *IEEE Transactions on Reliability*, vol. R-28, April 1979, pp. 58-61.
- [BLAK88] J. T. Blake, A. L. Reibman and K. S. Trivedi, "Sensitivity Analysis of Reliability and Performability Measures for Multiprocessor System", *Proc. 1988 ACM SIGMETRICS Int. Conference on Measurement and Modelling of Computer Systems*, Santa Fe, NM, 1988, pp. 177-186.
- [BEOU93] C. Beounes, M. Aguera, J. Arlat, S. Bachmann, C. Bourdeau, J.-E. Doucet, K. Kanoun, J.-C. Laprie, S. Metge, J. Moreira de Souza, D. Powell and P. Spiesser, "Surf-2: A Program for Dependability Evaluation of Complex Hardware and Software Systems", *Proc. FTCS 23*, IEEE Computer Society Press, 1993, pp. 668-673.
- [CARR86] J. A. Carrasco and J. Figueras, "Metfac: Design and Implementation of a Software Tool for Modelling and Evaluation of Complex Fault-Tolerant Computing Systems", *Proceedings FTCS 16*, IEEE Computer Society Press, 1986, pp. 424-429.
- [CIAR89] G. Ciardo, J. Muppala and K. S. Trivedi, "SPNP: Stochastic Petri Net Package", *International Conference on Petri Nets and Performance Models*, Kyoto, Japan, 1989, pp. 142-150.
- [CIAR90] 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.
- [CIAR91] G. Ciardo, J. Muppala and K. S. Trivedi, "On the Solution of GSPN Reward Models", *Performance Evaluation*, vol. 12, no. 4, 1991, pp. 237-254.
- [CIAR92] G. Ciardo, A. Blakemore, P. F. Jr. Chimento, J. K. Muppala and K. S. Trivedi, "Automated Generation and Analysis of Markov Reward Models using Stochastic Reward Nets", in *Linear Algebra, Markov Chains and Queueing Models*, Carl Meyer and Robert Plemmons (eds.), Springer-Verlag, 1992.
- [CLOS95] B. Muller-Closterman, "Numas - A Tool for the Numerical Analysis of Computer

- Systems", in D. Potier, ed., *Proceedings of the International Conference on Modelling Techniques and Tools for Performance Analysis*, North-Holland, 1986, pp. 141-154.
- [COUV91] J. A. Couvillion, R. Freire, R. Johnson, W. D. Obal II, A. Qureshi, M. Rai, W. H. Sanders and J. E. Tvedt, "Performability Modelling using UltraSAN", *IEEE Software*, September 1991, pp. 69-80.
- [FRAN95] R. G. Franks, A. Hubbard, S. Majumdar, J. E. Neilson, D. C. Petriu, J. A. Rolia and C. M. Woodside, "A Toolset for Performance Engineering and Software Design of Client-Server Systems", *Performance Evaluation*, vol. 24, no. 1-2, 1995, pp. 117-136.
- [FURC84] D. G. Furchtgott and J. F. Meyer, "A Performability Solution Method for Degradable Non-Repairable Systems", *IEEE Trans. on Computers*, vol. 33, no. 6, 1984, pp. 550-554.
- [GARG95] S. Garg, A. Puliafito, M. Telek and K. S. Trivedi, "Analysis of Software Rejuvenation using Markov Regenerative Stochastic Petri Net", in *Proceedings of 6th International Symposium on Software Reliability Engineering'95*, Toulouse, 1995.
- [GOYA86] A. Goyal, W. C. Carter, E. de Souza e Silva, S. S. Lavenberg and K. S. Trivedi, "The System Availability Estimator", *Proceedings FTCS 16*, IEEE Computer Society Press, 1986, pp. 84-89.
- [GOYA87a] A. Goyal and A. N. Tantawi, "Evaluation of Performability for Degradable Computer Systems", *IEEE Transactions on Computers*, vol. 36, no. 6, 1987, pp. 738-744.
- [GOYA87b] A. Goyal, S. S. Lavenberg and K. S. Trivedi, "Probabilistic Modelling of Computer System Availability", *Annual Oper. Research*, vol. 8, 1987, pp. 285-306.
- [GOYA87c] A. Goyal and S. S. Lavenberg, "Modelling and Analysis of Computer System Availability", *IBM Journal of Research and Development*, vol. 31, no. 6, 1987, pp. 651-664.
- [HARI87] 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.
- [HAVE89] B. R. Haverkort and I. G. Niemegeers, "Performability Modelling using Dynamic Queueing Networks", *ACM Performance Evaluation Review* 17(1), 1989, pp. 225.
- [HAVE90] B. R. Haverkort, "Performability Modelling Tools, Evaluation Techniques and Applications", Ph.D. Thesis, University of Twente, The Netherlands, 1990.

- [HAVE91] B. R. Haverkort and I. G. Niemegeers, "On the Mutual Performance-Dependability Influence in Dynamic Queueing Networks", in *Proc. International Workshop on Performance modelling of Computer and Communication Systems* (1991), pp. 33-40.
- [HAVE92] B. R. Haverkort, I. G. Niemegeers and P. V. van Zanten, "DyQNtool - A Performance Modelling Tool Based on the Dynamic Queueing Network Concept", in *Computer Performance Evaluation*, G. Balbo and G. Serazzi (Eds.), Elsevier Science Publishers B. V., 1992, pp. 181-195.
- [HAVE93a] 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.
- [HAVE93b] B. R. Haverkort, "Approximate Performability and Dependability Analysis using Generalized Stochastic Petri Nets", *Performance Evaluation*, vol. 18, 1993, pp. 61-78.
- [HAVE95] B. R. Haverkort, "Performability Modelling using DyQNtool, *Int. J. Reliability, Quality Safety Engineering*, vol. 2, no. 4, 1995.
- [HAVE96] B. R. Haverkort and I. G. Niemegeers, "Performability Modelling Tools and Techniques", *Performance Evaluation*, vol. 25, 1996, pp. 17-40.
- [HORO78] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Rockville, MD, 1978.
- [IBE89] O. C. Ibe, A. Sathaye, R. C. Howe and K. S. Trivedi, "Stochastic Petri Net Modelling of VAXcluster system Availability", *Proc. 3rd International Workshop on Petri Nets and Performance Models*, Kyoto, Japan, december, 1989, pp. 112-121.
- [JOGLE97] P. Jogalekar, "Scalability Analysis Framework for Distributed Systems", PhD Thesis, Dept. of Systems and Computer Engineering, Carleton University, OCIEE 97-10, 1997.
- [LEPO91] R. Lepold, "Performability Evaluation of a Fault-Tolerant Computer System using Stochastic Petri Nets", *Proc. 5th International Conference on Fault Tolerant Computing Systems*, Nurnberg, 1991.
- [LOGO95a] D. Logothetis and K. S. Trivedi, "The effect of detection and restoration times for a error recovery in communication networks", in *Proceedings of MILCOM*, 1995.

- [LOGO95b] D. Logothetis, K. S. Trivedi and A. Puliafito, "Markov Regenerative models", in *Int. Computer Performance and Dependability Symposium-IPDS95*, IEEECS Press, 1995.
- [LOPE93] N. Lopez-Benitez and K. S. Trivedi, "Multiprocessor Performability Analysis", *IEEE Transactions on Reliability*, vol. 42, No. 4, December 1993, pp. 579-587.
- [MCDO90] J. C. McDonald, *Fundamentals of Digital Switching*, Plenum Press, New York, 1990, Chapter 4.
- [MEYE80] J. F. Meyer, "On Evaluating the Performability of Degradable Computing Systems", *IEEE Trans. on Computers*, vol. 29, no. 8, Aug 1980, pp. 720-731.
- [MEYE82] J. F. Meyer, "Closed-Form Solutions of Performability, *IEEE Transactions on Computers*, vol. 31, no. 7, 1982, pp. 648-657.
- [MUPP91a] 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.
- [MUPP91b] J. K. Muppala, S. P. Woolet and K. S. Trivedi, "Real-time-systems Performance in the presence of failures", *IEEE Computer*, 1991, pp. 37-47.
- [MUPP92] 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.
- [NEIL95] J. E. Neilson, C. M. Woodside, D. C. Petriu and S. Majumdar, "Software Bottlenecking in Client-Server Systems and rendezvous Networks", *IEEE Transactions on Software Engineering*, vol. 21, no. 9, September, 1995, pp. 776-782.
- [NILS80] N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga Publishing Company, Palo Alto, California, 1980.
- [OBAL94] W. D. Obal II and W. H. Sanders, "Importance Sampling in UltraSAN", *Simulation* 62(2), 1994, pp. 98-111.
- [PAGE89] T. W. Page Jr. S. E. Berson, W. C. Cheng and R. R. Muntz, "An Object-Oriented Modelling Environment", *ACM Sigplan Notices*, vol. 24, no. 10 (*Proceedings OOPSLA'89*),

1989, pp. 287-296.

[RAI95] 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.

[REIB89] A. L. Reibman, R. Smith and K. S. Trivedi, "Markov and Markov Reward models transient analysis: An overview of Numerical Approaches", *European Journal of Operations Research*, vol. 40, 1989, pp. 257-267.

[ROLI95] J. A. Rolia and K. C. Sevcik, "The Method of Layers", *IEEE Trans. on Software Engineering*, vol. 21, no. 8, August 1995, pp. 689-700.

[SAHN86] R. A. Sahner and K. S. Trivedi, "A hierarchical, Combinatorial-Markov method of Solving Complex Reliability Models", *Proc. Fall Joint Computer Conference*, IEEE Computer Society Press, 1986, pp. 817-825.

[SAHN87a] R. A. Sahner and K. S. Trivedi, "Reliability Modeling using SHARPE", *IEEE Transactions on Reliability*, June 1987, pp. 186-193.

[SAHN87b] R. A. Sahner and K. S. Trivedi, "Performance and Reliability Analysis using Directed Acyclic Graphs", *IEEE Trans. on Software Engineering*, vol. 13, no. 10, October 1987, pp. 1105-1114.

[SAHN93] R. A. Sahner and K. S. Trivedi, "A Software Tool for Learning about Stochastic Models", *IEEE Trans. Ed.*, vol. 36, no. 1, 1993, pp. 56-61.

[SAHN96] R. A. Sahner, K. S. Trivedi and A. Puliafito (1996), *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package*, Kluwer Academic Publishers, Boston.

[SAND85] W. H. Sanders, A. Movaghar and J. F. Meyer, "Stochastic Activity Networks: Structure, Behavior and Application", *Proceedings of the International Workshop on Timed Petri Nets*, IEEE Computer Society Press, 1985, pp. 106-115.

[SAND87] W. H. Sanders and J. F. Meyer, "Performability Evaluation of Distributed Systems using Stochastic Activity Networks", *Proceedings of the 1987 International Workshop on Petri Nets and Performance Models*, IEEE Computer Society Press, 1987, pp. 111-120.

[SAND91] W. H. Sanders and J. F. Meyer, "Reduced Based Model Construction Methods for

Stochastic Activity Networks”, *IEEE Journal on Selected Areas Comm.*, vol. 9, no. 1, 1991, pp. 25-36.

[SMIT88] R. M. Smith, K. S. Trivedi and A. V. Ramesh, “Performability Analysis, Measures, an Algorithm and a case study”, *IEEE Trans. on Computers*, vol. 37, no. 4, 1988, pp. 406-417.

[SMIT89] R. M. Smith and K. S. Trivedi, “The Analysis of Computer Systems Using Markov Reward Processes”, in H. Takagi, ed., *Stochastic Analysis of Computer and Communication Systems*, Elsevier Science Publishers, 1989.

[TRIV90] K. S. Trivedi, O. Ibe, A. Sathaye and R. Howe, “Should I Add a Processor”, *23rd Annual Hawaii Conference on System Sciences*, January 1990, pp. 214-221.

[TRIV92] K. S. Trivedi, J. K. Muppala, S. P. Woollet and B. R. Haverkort, “Composite Performance and Dependability Analysis”, *Performance Evaluation* 14(1992), pp. 197-215.

[VEER91] 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.

[WOOD89] C. M. Woodside, “Throughput Calculation for Basic Stochastic Rendezvous Networks”, *Performance Evaluation*, vol. 9, no. 2, April 1989, pp. 143-160.

[WOOD95a] 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*, vol. 44, no. 1, January 1995, pp. 20-34.

[WOOD95b] C. M. Woodside, S. Majumdar, J. E. Neilson, D. C. Petriu, J. A. Rolia, A. Hubbard and R. B. Franks, “A Guide to Performance Modelling of Distributed Client-Server Software Systems with Layered Queueing Networks”, Technical Report SCE-95-23, Carleton University, Ottawa, Canada, November 1995.

[WOOD96] C. M. Woodside, “Performability modelling for multi-layered service systems”, *3rd International Workshop on Performability of Computer and Communications Systems*, BloomingDale, Illinois, Sept. 7-8, 1996.

Appendix A: BNF Description of FTLQNS Input File Format

This section gives the formal description of the FTLQNS input file grammar in BNF form. This grammar is an extension of the LQNS input file grammar provided in [WOOD95b]. The extensions have been highlighted. For the nonterminals the notation *<non-terminal_id>* is used, while the terminals are written without brackets as they appear in the input text.

The notation $\{\dots\}_n^m$, where $n \leq m$ means that the part inside the curly brackets is repeated at least n times and at most m times. If $n = 0$, then the part may be missing in the input text.

General Information

<ftlqnsInputFile> $\rightarrow$ *<generalInfo>* *<processorInfo>* *<taskInfo>* *<entryInfo>*

<generalInfo> $\rightarrow$ G *<comment>* *<convVal>* *<itLimit>* *<printInt>*_{opt}
*<underrelaxCoeff>*_{opt} *<endList>*

<comment> $\rightarrow$ *<string>* /* comment on the model */

<convVal> $\rightarrow$ *<real>* /* convergence value */

<itLimit> $\rightarrow$ *<integer>* /* max nb. of iterations */

<printInt> $\rightarrow$ *<integer>* /* intermed. res. print interval */

<underrelaxCoeff> $\rightarrow$ *<real>* /* underrelaxation coefficient */

<endList> $\rightarrow$ -1 /* end of list mark */

<string> $\rightarrow$ " *<text>* "

Processor Information

<processorInfo> $\rightarrow$ P *<np>* *<pDeclList>*

<np> $\rightarrow$ *<integer>* /* total number of processors */

$\langle pDeclList \rangle \rightarrow \{ \langle pDecl \rangle \}_0^{n_p} \langle endList \rangle$
 $\langle pDecl \rangle \rightarrow p \langle procId \rangle \langle schedulingFlag \rangle \langle procQuantum \rangle_{opt}$
 $\langle multiServerFlag \rangle_{opt} \langle replicationFlag \rangle_{opt}$
 $\langle procRate \rangle_{opt} \langle procFailProb \rangle$
 $\langle procId \rangle \rightarrow \langle integer \rangle \mid \langle identifier \rangle \text{ /* processor identifier */}$
 $\langle schedulingFlag \rangle \rightarrow f \text{ /* First come, first served */}$
 $\mid p \text{ /* Priority, preemptive */}$
 $\mid h \text{ /* Head of line */}$
 $\mid r \text{ /* Random */}$
 $\mid s \text{ /* Processor sharing */}$
 $\langle procQuantum \rangle \rightarrow \langle real \rangle \text{ /* processor time quantum */}$
 $\langle multiServerFlag \rangle \rightarrow m \langle copies \rangle \text{ /* number of duplicates */}$
 $\mid i \text{ /* Infinite server */}$
 $\langle replicationFlag \rangle \rightarrow r \langle copies \rangle \text{ /* number of replicas */}$
 $\langle procRate \rangle \rightarrow R \langle ratio \rangle \text{ /* Relative proc. speed */}$
 $\langle procFailProb \rangle \rightarrow (\langle real \rangle) \text{ /* Processor failure probability */}$
 $\langle copies \rangle \rightarrow \langle integer \rangle$
 $\langle ratio \rangle \rightarrow \langle real \rangle$

Task Information

$\langle taskInfo \rangle \rightarrow T \langle nt \rangle \langle tDeclList \rangle$
 $\langle nt \rangle \rightarrow \langle integer \rangle \text{ /* total number of tasks */}$
 $\langle tDeclList \rangle \rightarrow \{ \langle tDecl \rangle \}_0^{n_t} \langle endList \rangle$
 $\langle tDecl \rangle \rightarrow t \langle taskId \rangle \langle refTaskFlag \rangle \langle entryList \rangle \langle procId \rangle$

$\langle taskPri \rangle_{opt}$ $\langle thinkTimeFlag \rangle_{opt}$ $\langle multiServerFlag \rangle_{opt}$
 $\langle replicationFlag \rangle_{opt}$ $\langle taskFailProb \rangle$
 $\langle taskId \rangle \rightarrow \langle integer \rangle \mid \langle identifier \rangle$ /* task identifier */
 $\langle refTaskFlag \rangle \rightarrow r$ /* reference task */
 $\mid n$ /* non-reference task */
 $\langle entryList \rangle \rightarrow \{ \langle entryId \rangle \}_1^{ne_t} \langle endList \rangle$ /* task t has ne_t entries */
 $\langle entryId \rangle \rightarrow \langle integer \rangle \mid \langle identifier \rangle$ /* entry identifier */
 $\langle taskPri \rangle \rightarrow \langle integer \rangle$ /* task priority, optional */
 $\langle thinkTimeFlag \rangle \rightarrow z \langle real \rangle$ /* task think time, optional */
 $\langle taskFailProb \rangle \rightarrow (\langle real \rangle)$ /* task failure probability */

Entry Information

$\langle entryInfo \rangle \rightarrow E \langle ne \rangle \langle entryDeclList \rangle$
 $\langle ne \rangle \rightarrow \langle integer \rangle$ /* total number of entries */
 $\langle entryDeclList \rangle \rightarrow \{ \langle entryDecl \rangle \}_0^{ne \times 9} \langle endList \rangle$
/* k = maximum number of phases */
 $\langle entryDecl \rangle \rightarrow a \langle entryId \rangle \langle arrivalRate \rangle$
 $\mid c \langle entryId \rangle \{ \langle coeffOfVariation \rangle \}_1^k \langle endList \rangle$
 $\mid f \langle entryId \rangle \{ \langle phTypeFlag \rangle \}_1^k \langle endList \rangle$
 $\mid s \langle entryId \rangle \{ \langle serviceTime \rangle \}_1^k \langle endList \rangle$
 $\mid y \langle fromEntry \rangle \langle toEntry \rangle \{ \langle rendezvous \rangle \}_1^k \langle endList \rangle$
 $\mid ALT \ y \langle fromEntry \rangle [\langle toPrioEntryList \rangle]$
 $\{ \langle rendezvous \rangle \}_1^k \langle endList \rangle$
 $\langle arrivalRate \rangle \rightarrow \langle real \rangle$ /* open arrival rate to entry */

$\langle serviceTime \rangle \rightarrow \langle real \rangle$ /* mean phase service time */
 $\langle phTypeFlag \rangle \rightarrow \langle integer \rangle$ /* 0 - stochastic phase, 1 - deterministic phase */
 $\langle coeffOfVariation \rangle \rightarrow \langle real \rangle$ /* service time coeff. of variation */
 $\langle rendezvous \rangle \rightarrow \langle real \rangle$ /* mean number of RNVs/ph */
 $\langle fromEntry \rangle \rightarrow \langle entryId \rangle$ /* source of a message */
 $\langle toEntry \rangle \rightarrow \langle entryId \rangle$ /* destination of a message */
 $\langle toPrioEntryList \rangle \rightarrow \{ \langle entryId \rangle \}_1^{ne_{alt}} \langle endList \rangle$ /* ne_{alt} is the number of
alternative destinations of a message
which are prioritized from left to right */

Identifiers

Identifiers may be zero or more leading underscores ('-') followed by a character, followed by any number of characters, number of underscores. Punctuation characters and other special characters such as the dollar-sign ('\$') are not permitted.

Appendix B: Sample Intermediate Outputs generated by FTLQNS

This section provides the sample intermediate output data obtained by running FTLQNS on the model in Figure 4.4 whose model description file is ex414.ftlqn as shown in Figure 4.15. The sample output data corresponds to B, C, D, E and F of the flowchart in Figure 4.14. Figure B.1 illustrates a sample of generated fault tree shown in Figure 4.7(c) in SHARPE syntax. Figure B.2 illustrates the SHARPE's output for Figure B.1 from which the failure probability (highlighted) is extracted. Figure B.3 shows the *system activation states* and their probabilities generated in the file ex414.log by the AND-OR Graph Analyzer of the tool. Figure B.4 shows a sample of generated LQN model description file 4ex414.lqn corresponding to Figure 4.8(d). Figure B.5 shows the LQNS output file 4ex414.out corresponding to the LQNS input file 4ex414.lqn.

```
ftree ft
repeat t2 prob(0.08)
repeat p2 prob(0.05)
or e2 t2 p2
repeat t4 prob(0.04)
repeat p4 prob(0.09)
or e4 t4 p4
and ftRoot e2 e4
end
cdf(ft)
end
-1
```

Figure B.1 Fault Tree generated for Figure 4.7(c)

CDF for system ft:

probability at 0: 1.5926e-02
probability at infinity: 9.8407e-01
continuous probability: 0.0000e+00

Figure B.2 SHARPE's output for Figure B.1

In Figure B.2, the phrase *probability at infinity* implies the probability of occurrence of the failure of the system modelled by the fault tree.

Entries in a System Activation State (SAS):

[e1 e2 e3 e4 e5]

SAS: [1 0 1 0 1], probability: 0.010604, filename: 4ex414.lqn
SAS: [1 1 d 0 1], probability: 0.084118, filename: 3ex414.lqn
SAS: [1 0 1 1 d], probability: 0.084006, filename: 2ex414.lqn
SAS: [1 1 d 1 d], probability: 0.666559, filename: 1ex414.lqn

SAS: system failed state, probability: 0.154713

Figure B.3 System Activation States and their probabilities (generated file ex414.log)


```
G "generated for ex414.ftlqn" 0.000001 50 5 0.9 -1
```

```
P 3
```

```
p p1 f
```

```
p p3 f
```

```
p p5 f
```

```
-1
```

```
T 3
```

```
t t1 r e1 -1 p1
```

```
t t3 n e3 -1 p3
```

```
t t5 n e5 -1 p5
```

```
-1
```

```
E 3
```

```
s e1 0.000000 1.000000 -1
```

```
s e3 1.000000 1.000000 -1
```

```
s e5 1.000000 1.000000 -1
```

```
y e1 e3 0.600000 0.000000 -1
```

```
y e3 e5 0.800000 0.000000 -1
```

```
-1
```

Figure B.4 LQNS model input file 4ex414.lqn for Figure 4.8(d)

Copyright the Real-Time and Distributed Systems Group,
 Department of Systems and Computer Engineering
 Carleton University, Ottawa, Ontario, Canada. K1S 5B6

Generated by /usr/local/bin/lqns, version 23.2 (UNIX)

Input: 4ex414.lqn

Output: 4ex414.out

Comment: generated for ex414.ftlqn

Convergence test value: 6.24878e-07

Number of iterations: 8

MVA solver information:

Layer	n	step()	mean	stddev	wait()	mean	stddev
1	8	43	5.375	0.18298	699	87.375	6.0384
2	8	41	5.125	0.125	633	79.125	4.125
3	8	41	5.125	0.125	422	52.75	2.75
Total	24	125	5.2083	0.084681	1754	73.083	3.959

User: 0:00:00.01

System: 0:00:00.01

Elapsed: 0:00:00.00

Processor identifiers and scheduling algorithms:

Processor Name	Type	Copies	Scheduling
p1	Uni	1	FCFS (V phases)
p3	Uni	1	FCFS (V phases)
p5	Uni	1	FCFS (V phases)

Task information:

Task Name	Type	Copies	Processor Name	Pri	Entry List
t1	ref	1	p1	0	e1 (2 phases)
t3	serv	1	p3	0	e3 (2 phases)
t5	serv	1	p5	0	e5 (2 phases)

Figure B.5 LQNS output file 4ex414.out for input file 4ex414.lqn in Figure B.4

Entry execution demands:

Task Name	Entry Name	Phase 1	Phase 2
t1	e1	0	1
t3	e3	1	1
t5	e5	1	1

Mean number of rendezvous from entry to entry:

Task Name	Source Entry	Target Entry	Phase 1	Phase 2
t1	e1	e3	0.6	0
t3	e3	e5	0.8	0

Phase type flags:

All phases are stochastic.

Squared coefficient of variation of execution segments:

All executable segments are exponential.

Open arrival rates per entry:

All open arrival rates are 0.

Type 1 throughput bounds:

Task Name	Entry Name	Throughput
t1	e1	0.480769
t3	e3	0.357143
t5	e5	0.5

Mean delay for a rendezvous:

Task Name	Source Entry	Target Entry	Phase 1	Phase 2
t1	e1	e3	0.545455	0
t3	e3	e5	0.310804	0

Service times:

Task Name	Entry Name	Phase 1	Phase 2
t1	e1	1.55646	1
t3	e3	2.04864	1
t5	e5	1	1

Figure B.5 LQNS output file 4ex414.out for input file 4ex414.lqn in Figure B.4 (contd.)

Service time variance (per phase)
and squared coefficient of variation (over all phases):

Task Name	Entry Name	Phase 1	Phase 2	coeff of var **2
t1	e1	16.4007	1	2.6625
t3	e3	7.70971	1	0.937111
t5	e5	1	1	0.5

Throughputs and utilizations per phase:

Task Name	Entry Name	Throughput	Phase 1	Phase 2	Total
t1	e1	0.391166	0.608834	0.391166	1
t3	e3	0.2347	0.480816	0.2347	0.715516
t5	e5	0.18776	0.18776	0.18776	0.375519

Utilization and waiting per phase for processor: p1

Task Name	Pri n	Entry Name	Utilization	Ph1 wait	Ph2 wait
t1	0 1	e1	0.391166	0	0

Utilization and waiting per phase for processor: p3

Task Name	Pri n	Entry Name	Utilization	Ph1 wait	Ph2 wait
t3	0 1	e3	0.469399	0	0

Utilization and waiting per phase for processor: p5

Task Name	Pri n	Entry Name	Utilization	Ph1 wait	Ph2 wait
t5	0 1	e5	0.375519	0	0

Figure B.5 LQNS output file 4ex414.out for input file 4ex414.lqn in Figure B.4 (contd.)

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved