Fall 2005

Availability modeling and evaluation on high performance cluster computing systems

Hertong Song
Louisiana Tech University

NOTE TO USERS

AVAILABILITY MODELING AND EVALUATION ON HIGH PERFORMANCE CLUSTER COMPUTING SYSTEMS

by

Hertong Song, M.S.

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

COLLEGE OF ENGINEERING AND SCIENCE
LOUISIANA TECH UNIVERSITY

November 2005

LOUISIANA TECH UNIVERSITY

THE GRADUATE SCHOOL

November 1, 2005
Date

We hereby recommend that the dissertation prepared under our supervision by Hertong Song entitled

Availability Modeling and Evaluation on High Performance Cluster Computing Systems

be accepted in partial fulfillment of the requirements for the Degree of

Doctor of Philosophy in Computational Analysis and Modeling

Supervisor of Dissertation Research

Head of Department

CAM
Department

Recommendation concurred in:

Advisory Committee

Approved:
Director of Graduate Studies

Approved:
Dean of the Graduate School

Dean of the College

GS Form 13
(5/03)

ABSTRACT

Cluster computing has been attracting more and more attention from both the industrial and the academic world for its enormous computing power, cost effective, and scalability. Beowulf type cluster, for example, is a typical High Performance Computing (HPC) cluster system. Availability, as a key attribute of the system, needs to be considered at the system design stage and monitored at mission time. Moreover, system monitoring is a must to help identify the defects and ensure the system's availability requirement.

In this study, novel solutions which provide availability modeling, model evaluation, and data analysis as a single framework have been investigated. Three key components in the investigation are availability modeling, model evaluation, and data analysis. The general availability concepts and modeling techniques are briefly reviewed. The system's availability model is divided into submodels based upon their functionalities. Furthermore, an object oriented Markov model specification to facilitate availability modeling and runtime configuration has been developed. Numerical solutions for Markov models are examined, especially on the uniformization method. Alternative implementations of the method are discussed; particularly on analyzing the cost of an alternative solution for small state space model, and different ways for solving large sparse Markov models. The dissertation also presents a monitoring and data analysis framework, which is responsible for failure analysis and availability reconfiguration. In

addition, the event logs provided from the Lawrence Livermore National Laboratory have been studied and applied to validate the proposed techniques.

APPROVAL FOR SCHOLARLY DISSEMINATION

The author grants to the Prescott Memorial Library of Louisiana Tech University the right to reproduce, by appropriate methods, upon request, any or all portions of this Thesis. It is understood that "proper request" consists of the agreement, on the part of the requesting party, that said reproduction is for his personal use and that subsequent reproduction will not occur without written approval of the author of this Thesis. Further, any portions of the Thesis used in books, papers, and other works must be appropriately referenced to this Thesis.

Finally, the author of this Thesis reserves the right to publish freely, in the literature, at any time, any or all portions of this Thesis.

Author Hentong Song

Date Nov 10, 2005

GS Form 14
(5/03)

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

There are many people I want to thank for their help and encouragement during the period of my research work. First and foremost, I am grateful to my advisor, Dr. Chokchai Leangsuksun, for his direction, wise counsel and endless support through the course of this research; he helped me and taught me more than can be acknowledged in these few sentences. Thanks are also extended to Dr. Raja Nassar for his patient academic guidance and suggestions.

Many thanks to my committee, Dr. Weizhong Dai and Dr. Ben Choi, and CAM director Dr. Richard Greechie, for their support, guidance and helpful suggestions.

I would like to express gratitude to Steve L. Scotts and Christian Engelmann of Oak Ridge National Laboratory for their helpful comments and suggestions which contributed to my research work. I am in debt to Andy Yoo of Lawrence Livermore National Laboratory, who provided the valuable information for the research work to be sound.

My appreciation also goes to Professor William Stewart of North Carolina State University, Professor Joanne B. Dugan of University of Virginia, and Kishor. S. Trivedi of Duke University for their kindness academic support. I also appreciate the anonymous reviewers for their constructive criticism.

And finally, I wish to thank some of my friends, Yumin Zhang, Yingzong Bu, Li Wei, Xiangwei Zhao, Wei Long, Zhixin Huang, and the students at Louisiana Tech

University for their spiritual encouragement and support during this period. Thanks are also extended to Xiaodong Chen, Taiqing Qiu, Yuchen Qiu, Tingyu Lin, Xin Zhou and Zhengdong Chen for they are the pilot lights that always inspire me.

CHAPTER 1

INTRODUCTION

Cluster computing has become a cost effective and popular High Performance Computing (HPC) solution for its enormous computational power. High availability features have been increasingly vital to ensure that cluster computing environments can provide continuous services. It is imperative to know the dependability parameters during the conceptual design stages, since these parameters help in facilitating design trade-off and refinement (system cost vs. system reliability). The early evaluation of system characteristics, such as dependability [1], timeliness, and correctness, is necessary to assess whether the system being developed satisfies its goals and requirements. A typical availability modeling method is based on analytical formalisms such as fault tree [29][31], Markov chains [32][33][34], Stochastic Petri Net (SPN) [38][39], etc. This skill set requirement inevitably creates an issue for software designers, architects, and people in management who may be unfamiliar with these theoretical methodologies. Consequently, reliability engineers may be required to participate in the design and evaluation phases which lead to a two-step approach: system design and availability modeling. This situation clearly increases the complexity in team communication and, therefore, in product development. Moreover, the analytical models are still primitive; for example, a Markov chain consists of only state space and transitions, while SPN has

places and transitions. As a consequence, Markov chain and Petri Net models are often complex when the systems are non-trivial. These large models may be beyond the intuition of modelers, may lose the logic view of the system, and may become error prone.

The need for early evaluation demands standardized and well-defined design methods and languages. A variety of software packages exist to facilitate availability modeling and specify the models in compact forms [11][12][34][36][43]. The Unified Modeling Language (UML) is a widely-adopted standard modeling language used to visualize, specify, construct, and document the artifacts of a software system [2]. Using UML for availability modeling will reduce the gap between the system architects and software developers who are keen on using UML, but are unfamiliar with dependability modeling formalisms.

Once the cluster system is in operational stage, monitoring system's health is a must. This is to ensure that the system's runtime availability meeting its design goal, and helping to identify the trouble cause. Moreover, updating the system's availability information dynamically can provide most recent status of the system; which in turn enables the system's scheduler to make a better decision, thus improve the throughput of the cluster system.

The objective of this research is to investigate a novel technique that will facilitate the availability modeling, monitoring and evaluation on HPC cluster systems. Our approach is to create an availability model which is generic enough to simply the users' effort, eliminate the need for manual remodeling, and can be customized dynamically. Then, we exploit a systematic way towards generation and solution of the model.

Furthermore, the monitoring process should be able to dynamically update the availability information to reflect the most recent failure and repair events.

The outcome of this study will provide an effective solution for HPC cluster systems' availability modeling and evaluation. The research result will reduce the effort of availability modeling, provide most recent availability information, ensure the availability requirement, enable the scheduler to make a better decision, and finally will improve the overall system performance.

1.1 HPC Cluster System Architecture

Cluster computing technique involves clustering multiple commercial-of-the-shelf (COTS) computing nodes to accomplish performance and availability, which deals with perceptive and actual outages. With both hardware and software infrastructure components, clusters are aimed to achieve application workload sharing and fail-over capabilities. Among these, the Beowulf type cluster systems [44] have become popular for its price/performance, flexibility of configuration and update, and scalability.

A HPC cluster system consists of multiple computers communicated via network connections. The main HPC objective is normally targeted to achieve the best possible completion time and performance while executing a parallel application on as many possible nodes within the system. There are two types of nodes: head nodes (servers) and computing nodes (clients). Servers are taking requests and dispatch the tasks to the computing nodes, where the actual work processed. The systems are equipped with software packages to facilitate massive parallel computing; the MPI (message passing interface), for example, is a common programming paradigm. A parallel application runs simultaneously on a portion or all of the computers in the system. Unfortunately, if one

computer which the application is running on fails, the application hangs. This issue is currently a major drawback of MPI applications on the HPC cluster system.

Cluster systems can be configured based on the need [87][88][90]. For example, the servers can be classified as active-cold standby, active-warm standby, and active-active servers. Figure 1.1 illustrates an example of an HPC cluster system with two servers and multiple computing nodes. The active-active cluster means there are two (or more) servers are taking requests from the outside, mastering the cluster system. The active-warm standby type of cluster consists of a primary server, which is currently handling outside request, and the warm standby server is waiting to take over the control once the primary server has failed. The active-cold standby scheme is similar to the active-warm standby scheme; they differ in that once the primary fails, the cold standby server will be booted and take over the control.

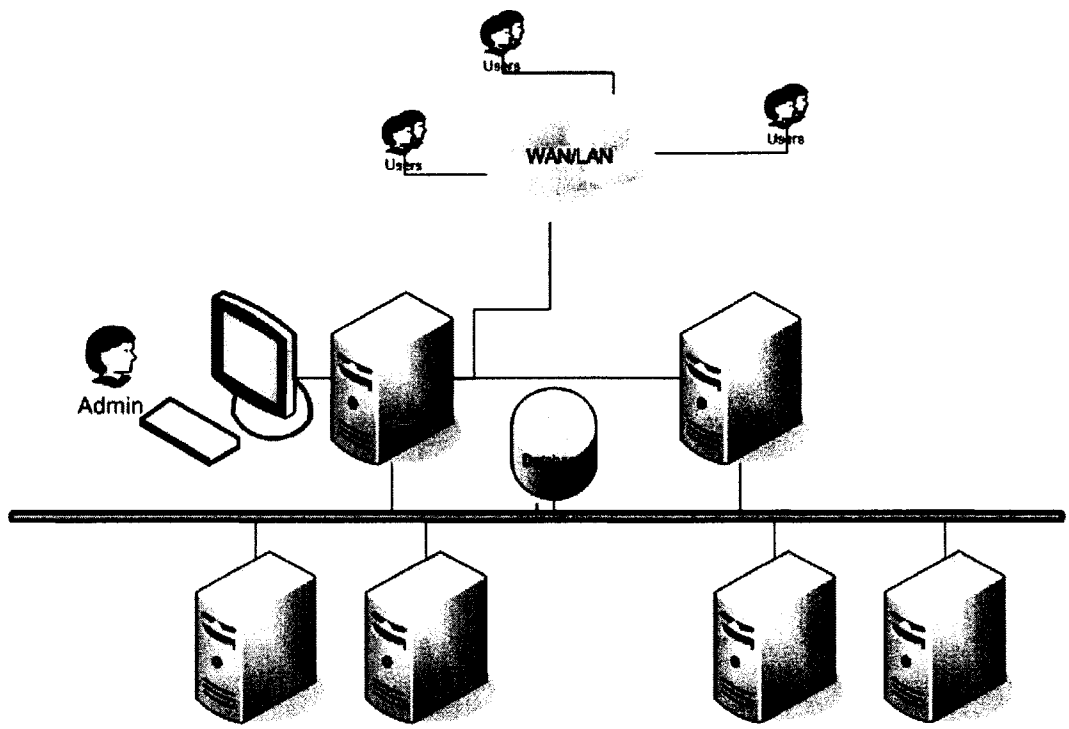

Figure 1.1 An HPC Cluster system with two servers

1.2 Overview of the Framework

This dissertation includes three major aspects, namely (1) the modeling, (2) model evaluation, and (3) monitoring and analysis. Figure 1.2 shows the overview of the framework included in this dissertation.

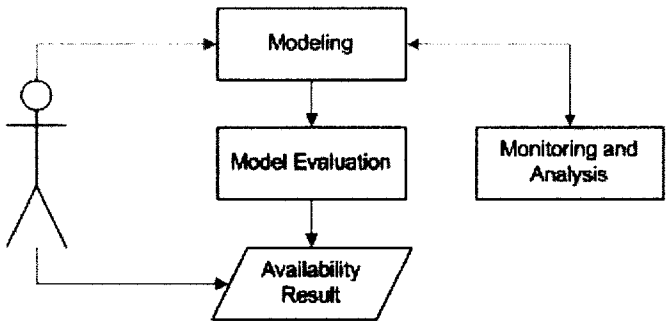

Figure 1.2 Overview of the framework

The modeling part investigates the possibility of using an alternative specification of the system availability model, including model decomposition. In this way, the availability model is more intuitive to the modeler, and is able to be updated during the runtime that facilitates the dynamic monitoring and analysis. An object-oriented modeling scheme [81][82][83] is studied for complex component interactions, and k-out-of-n structure is employed for independent identical components (i.i.d.) availability evaluation.

The modeling evaluation part is dealing with the solution of the availability models. Some models, such as the k-out-of-n structure, is given by formulas, while the others such as Markov model need numerical solutions. In this dissertation, the general methods of numerical solutions on Markov model are illustrated. This dissertation presents the uniformization method in detail for readers without much mathematical background, analyzes the possible alternative implementations and the cost of the small

state space implementation, and finally proposes a light weight solution on large sparse Markov models [84].

The monitoring and analysis is responsible for ensuring the system's health, performing data analysis, and updating the availability during the runtime. Currently, each computer in the system is assumed to be a single instance, and the failure and repair events are stored in the system's log file. The monitoring facility is responsible to extract these events of interest from the log file and write them into a maintained configuration file. Whenever there is a failure or a repair event happens, the monitoring facility will update the configuration file and the system's availability model [79]. Then it passes this information to the evaluation facility to reevaluate the system's availability. This framework can certainly be extended to monitoring applications of interest and record more detailed cause of the events. The latter needs to classify failures into more detailed events, group them into different categories in a tree like structure. This can also help to identify the root cause of the failure.

1.3 Components of the Framework

Each of the three major facilities in the framework includes several components. Figure 1.3 shows the components inside the framework in a tree like structure.

As stated earlier, the framework has three major facilities: the analytical modeling, the model evaluation, and monitoring and analysis. The modeling facility includes the modeling specification, in which the k-out-of-n structure and the object-oriented Markov model specification is adopted and developed. The UML modeling is given as an example and needs to be addressed in the future work.

The model evaluation facility includes the k-out-of-n structure and Markov model solutions. The formula for each component's availability is also given as the fundamental aspect in the k-out-of-n structure. The Markov model solution deals with the numerical solutions of Markov models. Different solution techniques are illustrated in the framework. The uniformization method is the major choice for solving Markov models. Alternative implementations of the method are discussed, particularly on the small state space and the way of solving large sparse Markov models. Runge-Kutta method is the second choice that can be used for the comparison purpose.

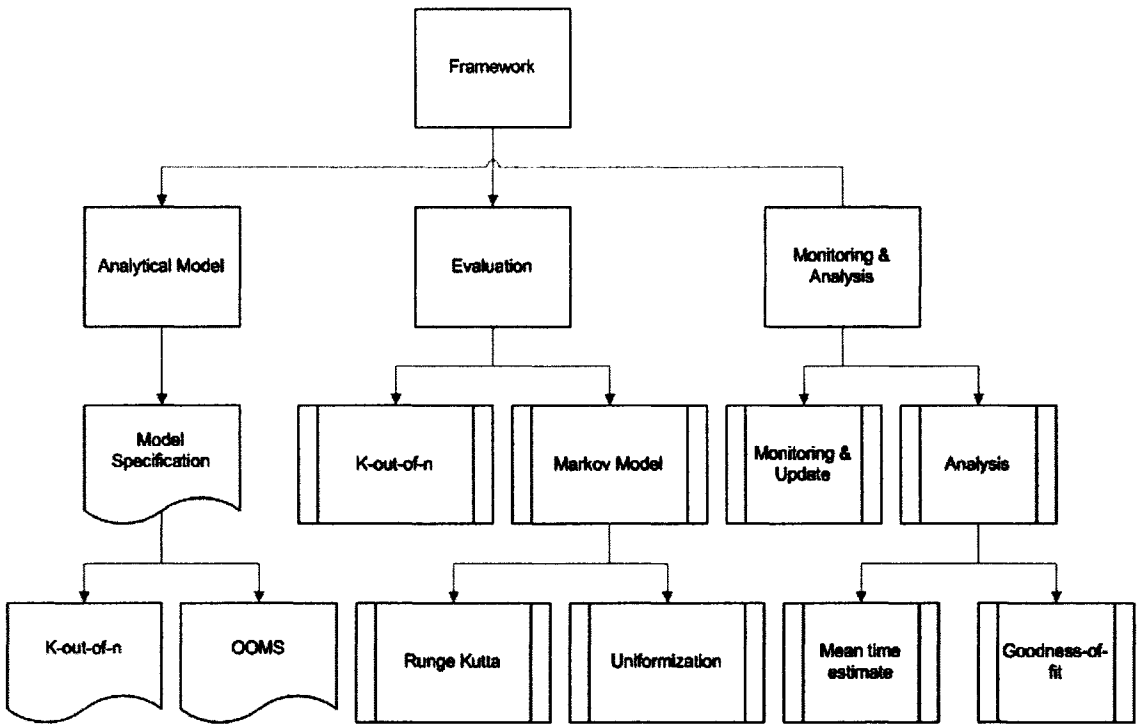

Figure 1.3 Components in the framework

The monitoring and analysis facility has two major functionalities, the monitoring and update daemon, and the data analysis feature. The first is used to monitor the system's health and update the system's availability configuration. The second is responsible for data analysis whenever there are failure and repair events. The data

analysis includes a mean time estimation and data fitting test. Currently, we propose using the goodness-of-fit technique to validate whether the events are fitted into exponential distribution. Other distribution such as Weibull distribution and alternative testing techniques such as Chapman-Smirnov test [71] may be included in the future.

1.4 Organization of the Dissertation

This dissertation is organized as follows: in Chapter Two, we introduce the background and general concepts of the reliability and availability modeling. In Chapter Three, we present the model decomposition and a variety of availability models. The object-oriented Markov model specification is illustrated in Chapter Four. Chapter Five depicts the numerical solution of Markov models. The monitoring and analysis facility is discussed in Chapter Six. The conclusion and future work is presented in Chapter Seven.

CHAPTER 2

BACKGROUND AND GENERAL CONCEPTS

In this chapter, we introduce some background information concerning the evaluation of vulnerability to fault tolerance systems. We begin by introducing some definitions of terminologies related to the measure of system vulnerability. Then we illustrate the three alternative measuring techniques, namely lifetesting, simulation, and analytical models. Since varieties of analytical models have been introduced in the past decades, a brief overview of the analytical models is depicted and categorized into combinatorial model, Markov models, and other models deviated from Markov models. We then explore the existing software packages for model generation and solutions. Finally, since the steady-state availability of a two-state Markov model is used through many parts of this dissertation, a closed form solution is given at the end of this chapter for reference.

2.1 Definition of Terminologies

Definition 2.1 Reliability

The reliability $R(t)$ of a system is the probability that the system survives until time t [31]. From a mathematical point of view, the reliability $R(t)$ of a system S can be expressed as:

$$R(t) = Pr(S \text{ is fully functioning in } [0, t])$$

Let X be the random variable representing the lifetime of a system, and let f be the probability density function (p.d.f.) and F be the cumulative density function (c.d.f.) of the variable X. Then the system reliability at time t can be depicted as

$$R(t) = \Pr(X > t) = 1 - F(t) = 1 - \int_0^t f(x)dx \tag{2.1}$$

Because

$$\int_0^\infty f(x)dx = 1$$

Hence, the reliability of a system can be expressed in

$$R(t) = \int_0^\infty f(x)dx - \int_0^t f(x)dx = \int_t^\infty f(x)dx \tag{2.2}$$

Normally, it is assumed that the system is working properly at the instant $t = 0$. Yet, it is possible to allow that the system is defective to begin with a probability p, i.e., $F(0) = p$. For such a case, the reliability of the system is

$$R(t) = 1 - p - \int_0^t f(x)dx \tag{2.3}$$

Definition 2.2 Availability

The instantaneous availability $A(t)$ of a system is the probability that the system is operating correctly at time t, regardless of the number of times it may have failed and been repaired in the interval $(0, t)$ [31].

The steady-state availability A_{ss} is a measure of the expected fraction time that the system is available for useful computation, and is obtained by taking the limit of $A(t)$ as $t \to \infty$, given that the limit exists. Thus,

$$A_{ss} = \lim_{t \to \infty} A(t) \tag{2.4}$$

Definition 2.3 MTTF

The mean time to failure (MTTF) of a system is the expected time until the occurrence of the system failure.

$$MTTF = \int_0^\infty R(t)dt \tag{2.5}$$

Definition 2.4 MTTR

The mean time to repair (MTTR) is a measure of expected time for repair of a failed system.

Definition 2.5 MTBF

The mean time between failure (MTBF) is a measure of expected mean time between failures in a system with repair, and it depends on both failure and repair processes, and

$$MTBF = MTTF + MTTR \tag{2.6}$$

If both the MTTF and MTTR are available, then the steady-state availability of the system may be calculated as:

$$A_{ss} = \frac{MTTF}{MTTF + MTTR} = \frac{MTTF}{MTBF} \tag{2.7}$$

From the definitions of reliability and availability given above, the difference between reliability and availability is that reliability requires that at no time within the interval $[0, t]$ may the system fail, whereas availability permits the possibility that the system may have failed and subsequently been repaired one or more times before time t.

In general, the type of system will determine whether reliability or availability is the preferred measure for a particular system. Reliability is required for non-repairable systems. Examples of such systems are flight control systems, safety control systems of

nuclear plants, and robots sent to Mars, for which repairs during the mission are impossible, or failures are disaster events. On the other hand, availability is usually preferred for repairable systems although reliability may also be a useful measure of interest. Examples of such systems are telecommunication switching systems and business transaction systems. In systems like these, operation often continues in a degraded state for a period of time following component failures. Laprie conceptualized dependability as a generic concept which covers a range of concepts including reliability, availability, safety, etc. [1]. As stated by Laprie, dependability is defined as the ability of the system to perform the task that is assigned to it or, more quantitatively, the probability that the system fulfills its tasks.

This dissertation considers the cluster computing systems which are repairable systems; therefore, availability is the primary measures of interest to us.

2.2 System Evaluation Techniques

When we are trying to evaluate the dependability of any systems, it is an attempt to predict the future safety behavior of the system based on historical information that is available about the system. The system vulnerability to failure evaluation techniques can be classified into three categories: lifetesting, simulation, and analytical models. This section briefly introduces the basic concepts on the three techniques.

2.2.1 Lifetesting

Lifetesting is to predict the system's behavior based on the existing technology, past experiences and observations. If the historical information of an existing component/system is available about its past performance and failure behavior, one may assume that it might continue to behave in the future as it has in the past. For example, if

a CPU chip is known for its failure rate, then we can assume that other chips manufactured with the same technology and under the same condition will exhibit the same failure rate. Therefore, lifetesting is the use of historical information to provide an estimate of future vulnerability to failure.

The usefulness of lifetesting is limited to systems that are constructed using well established technology, and it is not generally suitable for evaluating new systems constructed under the state of art technologies.

For highly reliable systems, failures of units in use are rare events. Hence, it might take years to get sufficient data to perform useful evaluations for such systems. Therefore, other methods must be considered to perform system evaluations, or for the purpose of aiding design decisions for systems that have not yet been built.

2.2.2 Simulation

Simulation is another technique to evaluate measures such as reliability, availability, and performance. The simulation process consists of three parts: (1) a computer representation of the relevant parts of the real world system is constructed first, (2) a series of random events to which the system must respond is generated to simulate the environment in which the system under test is embedded, and (3) observations are made of the system reaction to the events. This process is repeated numerous times in order to get statistically a number of trials at which the behavior of the model should closely resemble the behavior of the real system. Examples of simulation techniques can be found in studying the effect of a new generation bomb by simulating different surrounding environments.

Simulation is a useful modeling technique for evaluating measures because the computer representation can model the system to virtually any level of detail desired by the user. Furthermore, it is a versatile modeling technique, for it can be used to evaluate any measure for the system. The drawback is that the number of trials required may be large; therefore, the total computational effort required may be expensive.

2.2.3 Analytical Modeling

Analytical models are mathematical models which express particular aspects of system behavior that are of interest. A mathematical model is an abstraction from the real world system of aspects that relate only to the behavior and characteristics of the system that are of interest. All remaining details about the system are not considered, for the reason that managing the complexity of representing the system's behavior exactly in every detail is generally intractable. Furthermore, most details of the system's characteristics and behavior are not relevant to capturing the specific behavior of interest, so there seems little point to include them in the representation under evaluation.

Analytical models can sometimes give closed form results, but often they need to be solved using numerical techniques. Model size depends on both the number of components and the detail of modeled behaviors. The more components present within the system and the more detailed the system behavior, the larger the model size. When the model size becomes too large for the calculation of an exact result, suitable approximation techniques must be applied to the model to obtain acceptable result. Approximation techniques trade solution accuracy for model complexity, allowing the modeler to obtain a result within some range of values for a model which would otherwise be too large to be solved for any result at all.

Examples of analytical models are combinatorial models, Markov models, Petri Nets, and hierarchical models. Each type of these models requires different solution techniques and differs in the range of system behavior that they can model.

2.3 Analytical Models

Analytical models are mathematical models which are abstractions from the real world problems that relate only to the behavior and characteristics of the system that are of interest. Analytical models include reliability block diagram (RBD), fault tree (FT), reliability graph, Markov model, semi-Markov model, Petri Net, hybrid model, etc. [31]. In general, these models can be classified into two categories, namely the combinatorial models and Markov models. Reliability block diagram (RBD), fault tree (FT), and reliability graph are in the category of combinatorial models; the rest of the models mentioned above are in the Markov models category. The combinatorial models are also referred to as the non-state space models, and the Markov models are referred to as the state space models.

2.3.1 Combinatorial Models

Combinatorial models were the earliest type of analytical model in general use for system dependability analysis. Combinatorial models apply well to systems in which system failure behavior can be characterized by simple combinations of component failure. In general, combinatorial models capture the static behavior of the system. The solutions of these types are simply series-parallel reliability computations. Reliability block diagram (RBD), fault tree (FT), and reliability graph are in this category. They are similar in the way that they capture conditions that make a system fail in terms of structural relationships between the system components. In other words, they are visual

representations of so-called network reliability models. A fault tree without repeated components is equivalent to a reliability diagram, and both of them are a subset of the reliability graph, which is in turn a subset of the fault tree with repeated components.

The solution involves (1) a set of minimal paths (minpath) is generated, (2) all the paths are pair wise disjoint, and (3) apply the series-parallel formula. Algorithms for generating disjoint minpath are discussed in [15]-[19]. The major difference of these algorithms is that of using single variable inversion versus multiple variables inversion.

Reliability Block Diagram (RBD)

A reliability block diagram represents the logical structure of a system in regards to how the components' reliability affects the overall system reliability. Components are combined into blocks in series, in parallel or in k-out-of-n configurations. A series structure imposed on a set of components means that for the whole subsystem to work, every component has to be functioning. On the other hand, a parallel structure means that the whole subsystem can function if any one of the components is working. A k-out-of-n structure means that the whole subsystem can function if k or more of the components is working.

Figure 2.1 shows an example of a HPC cluster system with two servers and n computing nodes. The requirement to keep the system working is that at least one server and the computing nodes need to be functioning. Typically, in HPC environments, computing nodes model follow a series structure. On the other hand, the server model follows a parallel structure. The block diagram shows that the two servers are in parallel, and the n computing nodes in series.

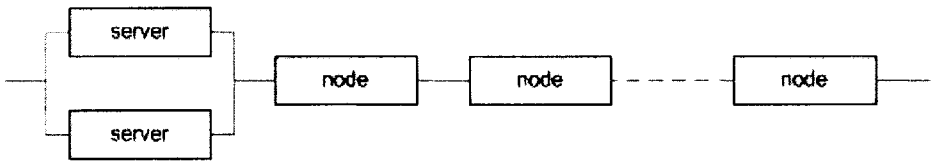

Figure 2.1 A RBD example

The distribution function for the failure time of a subsystem with n components is given by [29][31]:

$$F(t)=\begin{cases}\prod_{i=1}^{n}F_i(t) & \text{for a parallel structure}\\ 1-\prod_{i=1}^{n}(1-F_i(t)) & \text{for a series structure}\end{cases} \tag{2.8}$$

Where $F_i(t)$ is the probability for component i to fail at time t.

Fault Tree

Fault tree is a non-state space reliability modeling technique. A fault tree model represents all the sequences of individual component failures that cause the system to stop functioning in a tree like structure. The starting point is the root of the tree, which is the undesirable event of the system. A fault tree is a pictorial representation of the combination of events that can cause the occurrence of an undesired event. All of the events are combined by means of logic gates. Each gate has inputs and outputs. The input is either an event 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. On the other hand, the output of an OR gate is a logic 1 if and only if one or more of its input are at logic 1. Figure 2.2 shows the fault tree model of the two servers and n nodes cluster computing system as an illustrating example.

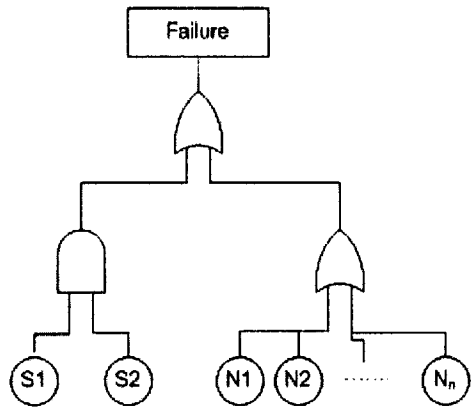

Figure 2.2 A Fault tree example

The failure distribution $F(t)$ for the failure time of a subsystem is computed as:

$$F(t)=\begin{cases}\prod_{i=1}^{n}F_i(t) & \textbf{AND}\ \text{gate}\\ 1-\prod_{i=1}^{n}(1-F_i(t)) & \textbf{OR}\ \text{gate}\end{cases} \tag{2.9}$$

Where $F_i(t)$ is the probability for component i to fail at time

The k-out-of-n structure

A k-out-of-n structure can be in two different forms, k-out-of-n "good", and k-out-of-n "fail". A k-out-of-n "good" structure means that the whole subsystem can function if k or more of the components is working. On the other hand, a k-out-of-n "fail" structure means that the whole subsystem fails if k or more of the components has failed. The backbone of the k-out-of-n structure is in a form of binomial trials. Thus, the failure distribution $F(t)$ for the failure time of a subsystem for a k-out-of-n "fail" structure is

$$F_{k|n}(t)=\sum_{i=k}^{n}\binom{n}{i}F(t)^i(1-F(t))^{n-i} \tag{2.10}$$

The distribution function for a k-out-of-n "good" structure can be acquired in the same way. Note that, Equation 2.10 is applied to a k-out-of-n "fail" structure with n

identical components. If the structure constitutes non-identical components, the corresponding equation is complicated. Varieties of algorithms [13][14] exist to make the calculations faster on this behave, and they are out of the scope of this dissertation.

Reliability Graphs

A reliability graph is another representation of combinatorial models. It consists of a set of nodes and edges. The graph contains a source node and a destination node, and the edges are assigned with failure probability or failure rates. A system represented by a reliability graph fails when there is no path from the source node to the destination node. The reliability or unreliability of the graph is solved by (1) generating the set of minpath or mincut [15]-[20], (2) making all the path or cut pair wise disjoint, and (3) summing up the probability of all the minpath or mincut.

Availability Modeling Using Combinatorial Models

Combinatorial models cannot capture the repair event. For this reason, they are generally used for reliability measures. However, with certain modifications, and assume each components are stochastic independent, combinatorial models can be applied to system availability measures.

For a repairable component with failure rate λ and repair rate μ, its instantaneous availability $A(t)$ is

$$A(t) = \frac{\mu}{\mu+\lambda} + \frac{\lambda}{\mu+\lambda} e^{-(\mu+\lambda)t} \tag{2.11}$$

and its instantaneous unavailability $U(t)$ is

$$U(t) = 1 - A(t) = \frac{\lambda}{\mu+\lambda} - \frac{\lambda}{\mu+\lambda} e^{-(\mu+\lambda)t} \tag{2.12}$$

The proof is given at the end of this chapter, via a two state Markov model. Together with the equations shown above, the combinatorial model can be applied to availability modeling, under the assumption that the components are stochastic independent.

2.3.2 Markov Models

Dependability models often need to capture the sequence of component failures when modeling fault tolerant systems. Combinatorial models have difficulty in capturing this type of system behavior because of their combinatorial characteristics. To address this type of system behavior, Markov models have been popular techniques applied to a variety of fields. A Markov process, which is a special case of stochastic processes whose dynamic behavior is such that, the probability distributions for its future development depend only on the present state and not how the process arrived in that state. From the view of a system model, a Markov process represents the system as a finite group of states in which the system can exist, and a set of transitions that moves the systems between states over time. The following gives the general concepts.

Definition Stochastic Process

A stochastic process is a family of random variables $\{X(t) | t \in T\}$, defined on a given probability space, indexed by the parameter t, where t varies over an index set T.

Definition Markov Process

A stochastic process $\{X(t) | t \in T\}$ is called a Markov process if for any $t_0 < t_1 < \cdots < t_n < t$, the conditional distribution of $X(t)$ for given values of $X(t_0), X(t_1), \cdots X(t_n)$ depends only on $X(t_n)$, that is

$$P[X(t) \le x \mid X(t_n) = x_n, X(t_{n-1}) = x_{n-1}, \cdots, X(t_0) = x_0] = P[X(t) \le x \mid X(t_n) = x_n] \quad (2.13)$$

A Markov process is called a Markov chain if the state space is discrete. If the parameter time space T of a Markov chain is discrete, then it is a discrete time Markov chain (DTMC). If the parameter space T is continuous, it is a continuous time Markov chain (CTMC), otherwise. In general, a Markov process is referred to as a continuous time Markov chain. If the transition rates are constant, the Markov chain is said to be homogeneous. This implies that the time spent waiting in any state is exponentially distributed. On the other hand, if the rates of transitions are functions of time instead of constants, the Markov chain is said to be non-homogeneous. If the distributions of the holding time of any states are general distributions other than exponential distribution, the Markov model is said to be a semi-Markov model.

Classification of States

- The state j is said to be reachable from state i if the probability $p_{ij} > 0$.
- Transient state: A state is said to be transient if and only if there is a positive probability that the process will not return to this state.
- Recurrent state: A state is said to be recurrent if and only if, starting from a state, the process eventually returns to the state with probability one.
- Periodic and aperiodic state: A recurrent state is periodic with period d if $p_{ij}(t) = 0$ except $t = d, 2d, \cdots$. A state that is not periodic is an aperiodic state.
- Absorbing state: A state i is said to be an absorbing state if and only if $p_{ii} = 1$
- Ergodic state: An aperiodic recurrent state with a finite mean recurrent time is called ergodic.

The majority Markov models of repairable computing systems are ergodic. In this dissertation, only homogenous ergodic CTMC is of interest.

Continuous Time Markov Chain [50]

Let E be a countable set of state space, and let $\{X(t) \mid t \geq 0\}$ stochastic process $t \in R^+$ and $X(t) \in E$. $\{X(t) \mid t \geq 0\}$ is called a continuous time Markov chain if $\forall i, j, i_1, i_2, \cdots, i_k \in E$, $\forall t, s \in R^+$, and $\forall s_1, s_2, \cdots, s_k \in R^+$ with $s_l \leq s$ for all $l \in [1, k]$,

$$P[X(t+s) = j \mid X(s) = i, X(s_1) = i_1, \cdots, X(s_k) = i_k] = P[X(t+s) = j \mid X(s) = i] \quad (2.14)$$

whenever both sides are well defined. It is called homogeneous if Equation 2.14 is independent of s.

Chapman-Kolmogorov Equation

Let P be the probability transition matrix of a CTMC, and $P = \{p_{ij}(t)\}$, where $i, j \in E$ and $p_{ij}(t) = P[X(t+s) = j \mid X(s) = i]$. The Chapman-Kolmogorov Equation can be described as

$$P(t+s) = P(t)P(s) \quad (2.15)$$

with the properties of $P(0) = I$ and $P(t)e = e$, where I is the identify matrix and e is a vector with all elements are 1s.

Chapman-Kolmogorov forward differential equation

The infinitesimal generator matrix Q is defined as

$$Q = \lim_{\Delta t \to 0} \left\{ \frac{P(t, t+\Delta t) - I}{\Delta t} \right\} \quad (2.16)$$

and the Chapman-Kolmogorov forward differential equation is defined as

$$\frac{dP(t)}{dt} = P(t)Q \quad (2.17)$$

Let $\pi(0)$ and $\pi(t)$ be the initial probability vector and the probability vector of the CTMC at time t, respectively ($\pi(t) = \pi(0)P(t)$), then Equation 2.15 can be expressed as

$$\frac{d\pi(t)}{dt} = \pi(t)Q \tag{2.18}$$

A stationary distribution of continuous time Markov chain is any probability vector π on the state space E such that $\forall t \in R^+$

$$\pi(t)Q = 0 \tag{2.19}$$

and

$$\pi e = 1 \tag{2.20}$$

where e is a vector with all elements are 1s, i.e., $e = \{1,1,\cdots,1\}$.

2.3.3 The Other Markov Models

This section briefly introduces some models deviated from the Markov models, include Markov reward model, Petri Nets, and Hybrid models. Because of the fact that they are built on top of the Markov model, therefore, it is reasonable to place them into the Markov models category.

Markov reward model [35] is a Markov model with the reward assigned to each states and transitions, and each submodel is linked by mathematical expressions.

Petri Nets [40] consists of places and transitions, and it is more recent than other models. A number of tokens exist in the net and migrate from place to place according to the rules upon which each transition becomes enabled. Stochastic Petri Nets permit the transitions to require a delay period that has a specified distribution before the transition becomes enabled. Petri nets can model system behaviors such as event conflicts,

concurrency, sequencing, forking, joining, and synchronization. Petri net models may be evaluated either by simulation or by reduction to a Markov model. There are several kinds of nets, such as generalized stochastic Petri nets (GSPN) [39], stochastic reward nets (SRN) [41], etc. [38].

Hybrid models are analytical models that integrate two or more different types of analytical models together, often in a hierarchical way. The software packages that support hybrid models include CARE [21], HARP [22], and SHARP [23].

Dynamic Fault Tree (DFT) uses the fault tree representation to capture systems dynamic behaviors by adding several logical gates [42], such as functional dependency gate, sequence enforcing gate, priority AND gate, etc. A software package is necessary to host a DFT model, and translate the model into an underlying Markov model. The Markov model is then solved and the result is given to the modeler.

2.4 Model Generation and Existing Software Packages

System modeling and analysis is generally a two-step process: (1) abstracting the real world problem into an analytical model, and (2) solving the model to produce a value for the measure of interest. The first step is to create the model, and it is the task of modelers; the second step is almost always solved by using a computer, except for trivial problems which can be solved by hand. Thus, analytical modeling is a technique of choosing the best way to represent the model (the modeling part), to solve the model (the mathematical part). One can also argue that the analytical modeling is a three step process: create the model, represent the model into a computer, and solve the model by the computer. In this case, the modeler needs to choose the model type, abstract the system into the model, and represent the model in the computer.

Creating an analytical model needs the modeler to intimately understand the system and its behavior, and be well versed in the underlying mathematical theories. Experience is the most effective way to increase the skill. It often requires step learning curve in one area (the system) or the other (mathematical theory) by a person doing the modeling work. Once the system and the underlying mathematics are understood, the modeler then needs to capture the relevant behavior of the system into the model. Modeler should consider tradeoffs between complexity and accuracy. The more details and more components captured in the model, the larger and more complex the model will be. As a consequence, it will take a long time to have a solution, or there is no solution for the reason that computer cannot handle such a model. If the model is too complex to be solved by computers, the modeler needs either to simplify the model by eliminating more detail behavior of the system such as model decomposition at system level, coverage modeling, or use mathematical theories to have a approximate solution such as state space aggregation and truncation. Certainly, the system can be modeled in one way or another, and an approximation is better than no solution at all.

Therefore, the analytical modeling becomes an iterative process, that is, representing the system in a model, solving the model by a computer. If the solution is not satisfied, the modeler needs to go back to remodel the system until a desired solution is achieved. Sometimes the correctness of the solution cannot be guaranteed, because the system can be modeled in many ways, and the computer may not always give the right result due to the algorithm applied and the round off errors. One way to get a good feeling on the model is for the modeler to build up different types of models, and/or to

use a completely different computer package to evaluate the exact same measure and compare the results.

There are a variety of software packages that exist [11][12] to facilitate models solving. Some are programs oriented, and some of them are equipped with GUI editors to provide modelers with visual means to enter the model into the computer. CARE [21], HARP [22], and SHARPE [23] support combinatorial modeling. Dependability modeling software packages that solve continuous time homogeneous Markov chains include ARIES, SURF, SAVE, and ACE [11]. CARE and HARP support solving homogeneous and non-homogeneous Markov models. HARP, SHARPE and SURE are also designed to solve semi-Markov models. There are many software tools available for stochastic Petri net specification and analysis, which include SPNP [24], GreatSPN [25], ESP [26], and UltraSAN [27]. Galileo [28] is the software package for dynamic fault tree modeling.

2.5 Two State Markov Model

This section gives the proof to the availability of a repairable component via a two states Markov model. This concept is used to achieve availability modeling via combinatorial models, and will be used in this dissertation later on. Thus, it is necessary to lay out the proof herein.

Considering a repairable component with failure rate λ and repair rate μ, Figure 2.3 shows the corresponding two-states Markov model of the component. The model has two states, state 1 and state 0. State 1 means the component is functioning, and state 0 means the component has failed.

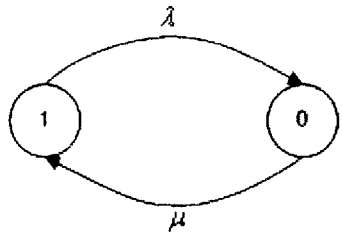

Figure 2.3 A two-state Markov model

The availability of the component is actually the probability that it is in state 1. The generator matrix is:

$$Q=\begin{bmatrix} -\mu & \mu \\ \lambda & -\lambda \end{bmatrix} \tag{2.21}$$

Then the differential equation becomes

$$\begin{aligned} \pi_0' &= -\mu\pi_0 + \lambda\pi_1 \\ \pi_1' &= \mu\pi_0 - \lambda\pi_1 \end{aligned} \tag{2.22}$$

Note that π_0 is used here instead of $\pi_0(t)$, and π_1 instead of $\pi_1(t)$ for the purpose of clarity. Because $\pi_0 + \pi_1 = 1$, so the second equation of (2.22) can be written as

$$\pi_1' = \mu(1-\pi_1) - \lambda\pi_1 \tag{2.23a}$$

or

$$\pi_1' + (\mu+\lambda)\pi_1 = \mu \tag{2.23b}$$

Multiply both sides by $e^{\int(\mu+\lambda)dt} = e^{(\mu+\lambda)t}$, we get

$$\begin{aligned} e^{(\mu+\lambda)t}\pi_1' + (\mu+\lambda)e^{(\mu+\lambda)t}\pi_1 &= \mu e^{(\mu+\lambda)t} \\ \left(e^{(\mu+\lambda)t}\pi_1\right)' &= \mu e^{(\mu+\lambda)t} \\ \left(e^{(\mu+\lambda)t}\pi_1\right) &= \frac{\mu}{\mu+\lambda}e^{(\mu+\lambda)t} + c \\ \pi_1 &= \frac{\mu}{\mu+\lambda} + ce^{-(\mu+\lambda)t} \end{aligned} \tag{2.24}$$

use the initial condition $\pi_1(0)=1$, we get $c=\frac{\lambda}{\mu+\lambda}$. Then the availability of the component at time t is

$$A(t)=\pi_1(t)=\frac{\mu}{\mu+\lambda}+\frac{\lambda}{\mu+\lambda}e^{-(\mu+\lambda)t} \tag{2.25}$$

CHAPTER 3

SYSTEM AVAILABILITY MODELING

3.1 Overview of the Approach

In general, this dissertation deals with modeling availability of a HPC cluster [44][45] using familiarized notations, such as UML, as the front end design representation [89]. The approach is to embed the statistical parameters and information regarding properties that affect the system availability. Then, it exploits these designs in a systematic way towards generation of models that serve as input for reliability and availability evaluation. The approach requires minimum user effort and eliminates the need for manually remodeling of the system since the user can change the design in the UML front end. Figure 3.1 illustrates our integrated UML availability modeling framework.

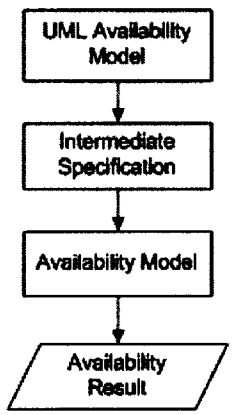

Figure 3.1 Modeling framework

The UML based model incorporating the information needed for dependability analysis is transformed into an intermediate model which adopts an object-oriented specification. Then, the intermediate model is transformed into the analytical model.

In this approach, the intermediate model representation is to mimic the UML class and statechart diagrams; thus, it is a one-to-one mapping from the UML model. Currently, we have implemented part of the intermediate model, which is represented in Chapter Four. This chapter focuses on presenting alternative availability models for cluster computing systems. It first presents the model decomposition approach, and then describes several possible availability models and their availability estimation.

3.2 Model Decomposition

A HPC cluster system consists of a server and a set of client nodes. The server takes requests and forwards the jobs to a subset of clients, while the clients handle the jobs and respond back to the server once the work is done. The server and clients are linked together through a network connection. If the server fails then the whole system is down; thus, the system suffers from a single point failure problem. The HA-OSCAR [47] introduced a standby server to the system in addition to the primary server.

The HA-OSCAR has a primary server and a warm-standby server. The primary server provides the services and processes all the user's requests. The standby server is waiting to take control when a failure in the primary server is detected. When the primary server fails, after a certain time, the monitoring facility will detect the failure, and the standby server will "wake up" and take over the control. Once the primary server gets repaired, it will take back control of the system, and put the standby server back to "dormant". More servers can be added into the system with different configurations to

reduce the downtime, hence, increase the availability of the system.

The availability of the cluster system is assessed normally when there is at least a server and a quorum of clients functioning. If there are many processors involved in the system, and if the continuous time Markov chain model is chosen to describe the system's dependability, then the resulting availability model could be extremely large. Thus, we adopt the "hierarchical composition" technique [29][30][48][49]. The system availability model is divided into two submodels based on the functionalities of subsystems, a server submodel and a client submodel. The availability model can be described by using a RBD (Reliability Block Diagram), as shown in Figure 3.2.

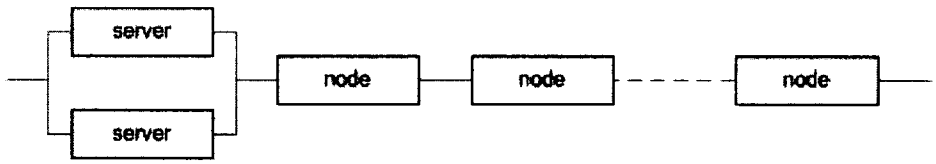

Figure 3.2 The RBD of system availability model

The system fails when either the server submodel or the client submodel fails. Since $A = 1 - F$, from Equation 2.8, we have $A(t) = \prod_{i=1}^{n}(1 - F_i(t)) = \prod_{i=1}^{n} A_i(t)$. Hence, the system's availability is:

$$A_{sys} = A_S A_N \tag{3.1}$$

where A_{sys}, A_S, and A_N denote the availability for the system, server, and client model, respectively. For the simplicity purpose of availability evaluation, we assume that the N client nodes are identical and exponentially distributed with failure rate λ_n and repair rate μ_n. The client submodel requires at least N client nodes to keep the system functioning. The availability of the client submodel is given by

$$A_N = \prod_{i=1}^{N} A_n^i \tag{3.2}$$

where A_n and $\overline{A}_n$ are the availability and unavailability of a single client node at time t, given by [37]

$$A_n(t) = \frac{\mu_n}{\lambda_n + \mu_n} + \frac{\lambda_n}{\lambda_n + \mu_n} e^{-(\lambda_n + \mu_n)t} \tag{3.3}$$

$$\overline{A}_n(t) = 1 - A_n(t) \tag{3.4}$$

On the other hand, servers are normally implemented with complex mechanisms, leading to a complicated model, hence need to describe more intricate interactions. The complicated behavior can be modeled by a continuous time Markov chain [29][30][31][32]. We adopt an object-oriented, event generating, and message passing technique [36] to specify the interaction between servers. The object-oriented specification of the servers and the transformation that converts the specification into a corresponding continuous time Markov model are described in Chapter Four.

3.3 Alternative of Availability Models

This section lists a few alternative availability models based on the cluster system configuration. The server's availability models are specified by Markov models with states and transitions listed. The functioning states are marked as Y. We assume the servers have the same failure and repair rates, and the system is functioning if there is at least one server working. Figure 3.1 lists some parameters that will be used for the availability estimation.

Table 3.1 Parameters for availability estimation

Input Parameters	Mean Time	Rate
Primary server failure	5,000 hrs	0.0002
Primary server repair	4 hrs	0.25
Failure detection	5 min	720
Warm standby server failure	5,000 hrs	0.0002
Active standby server failure	10,000 hrs	0.0001
Standby server activation	5 min	720
Standby server deactivation	5 min	720
Standby server repair	4 hrs	0.25

Single Active Server. The Markov model for a single active server is a simple two state model, and it is given by the end of Chapter Two. Thus, the model is omitted here.

Active-Standby Server. The Markov states and transitions of the active-standby servers are given in Chapter Four as an example; thus, the model is not repeated here.

The k-out-of-n Structure. The k-out-of-n structure is based on Formula 2.10.

Active-Active Servers. The Markov states and transitions of the active-active servers are listed in Table 3.2.

Three Active Servers. The Markov states and transitions of three active servers are listed in Table 3.3.

Table 3.2 Active-Active Markov states and transitions

#	States		
0:	U	U	Y
1:	D	U	Y
2:	U	D	Y
3	D	D	

#	Source	Destination	Rate
1	0	1	0.0002
2	0	2	0.0002
3	1	0	0.25
4	1	3	0.0002
5	2	3	0.0002
6	2	0	0.25
7	3	2	0.25
8	3	1	0.25

Table 3.3 Three active servers

#	states	
0	UUU	Y
1	DUU	Y
2	UDU	Y
3	UUD	Y
4	DDU	Y
5	DUD	Y
6	UDD	Y
7	DDD	

#	Source	Destination	rate
1	0	1	0.0002
2	0	2	0.0002
3	0	3	0.0002
4	1	0	0.25
5	1	4	0.0002
6	1	5	0.0002
7	2	4	0.0002
8	2	0	0.25
9	2	6	0.0002
10	3	5	0.0002
11	3	6	0.0002
12	3	0	0.25
13	4	2	0.25
14	4	1	0.25
15	4	7	0.0002
16	5	3	0.25
17	5	7	0.0002
18	5	1	0.25
19	6	7	0.0002
20	6	3	0.25
21	6	2	0.25
22	7	6	0.25
23	7	5	0.25
24	7	4	0.25

3.4 Availability Estimations

In this section, we list some figures of the evaluation results for the availability models presented in the previous section, as shown in Figure 3.3 - Figure 3.7. The parameters used are given in Table 3.1. We can see that for a single server, it reaches the steady-state availability at 0.9992. Both of the active-active and active-standby servers reach the six-nine (0.999999). The three active servers reach 9-nine.

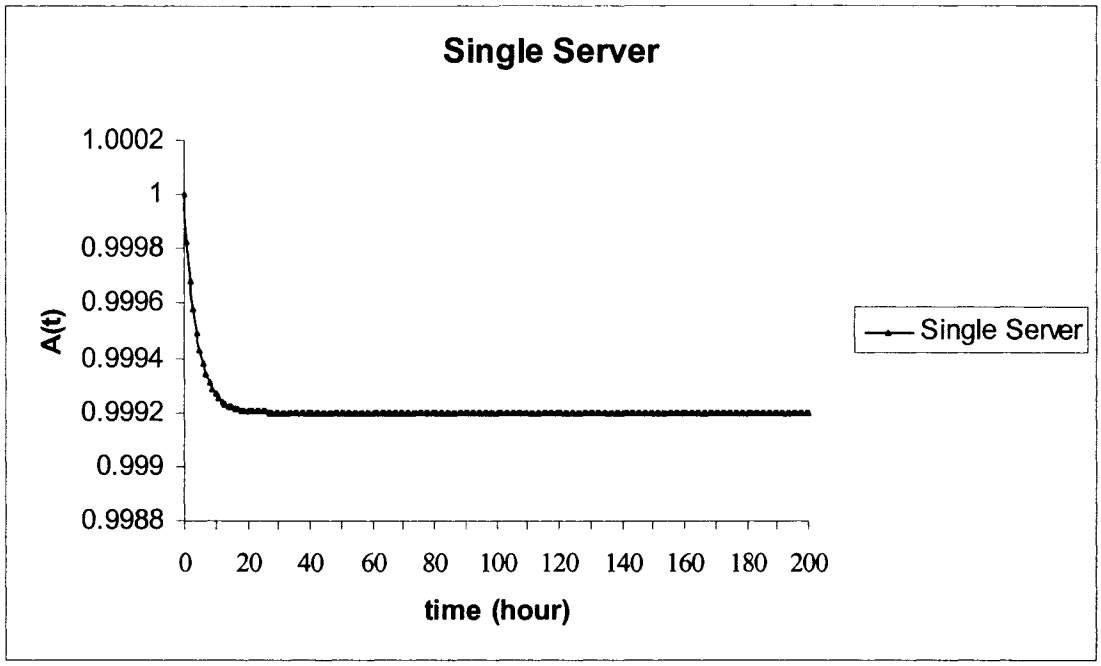

Figure 3.3 Single server availability estimation

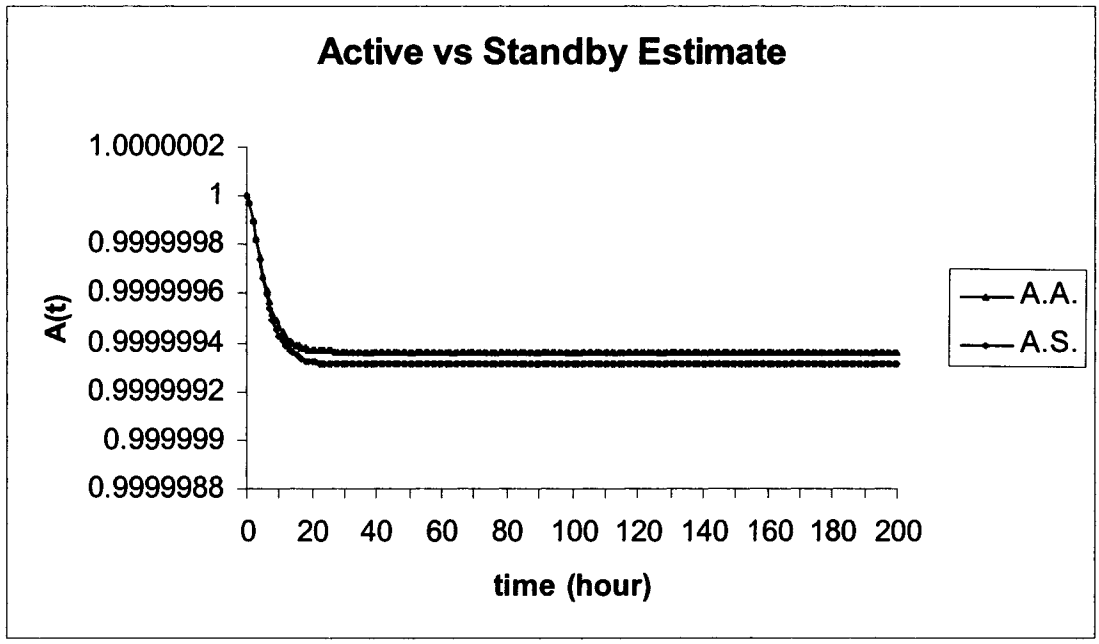

Figure 3.4 Active vs standby server availability estimation

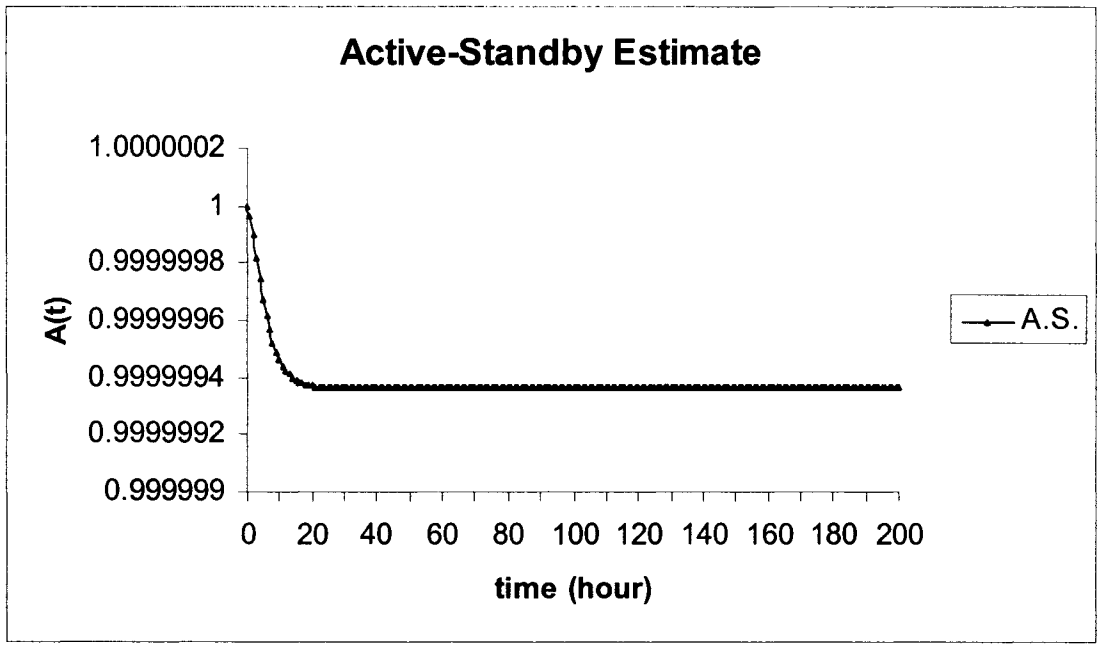

Figure 3.5 Active-Standby server availability estimate

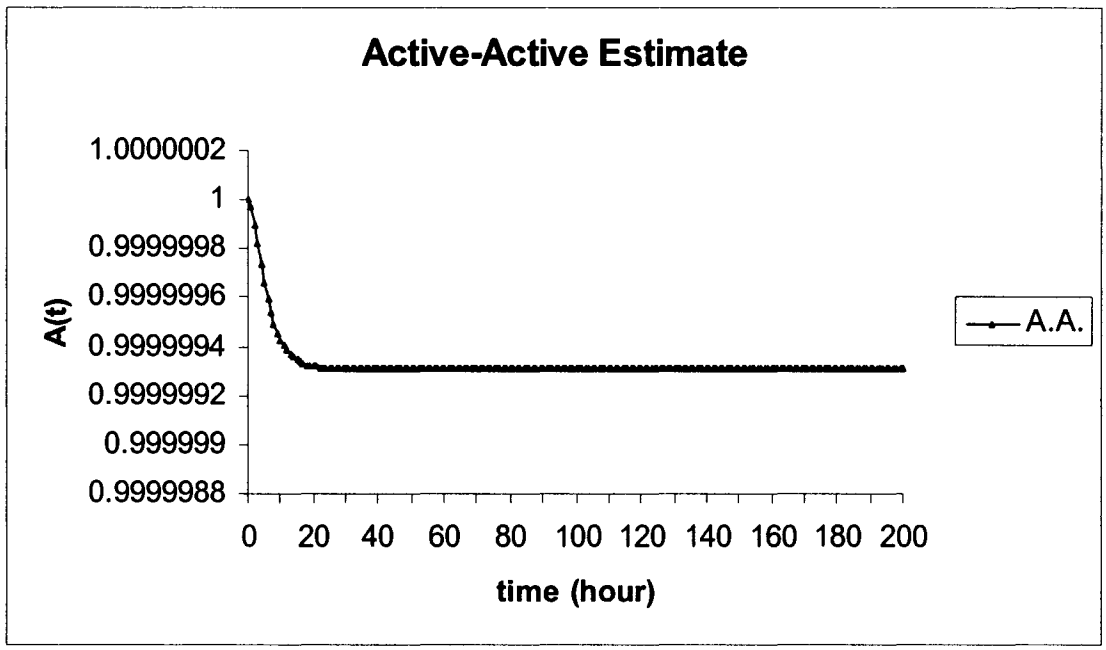

Figure 3.6 Active-Active server availability estimate

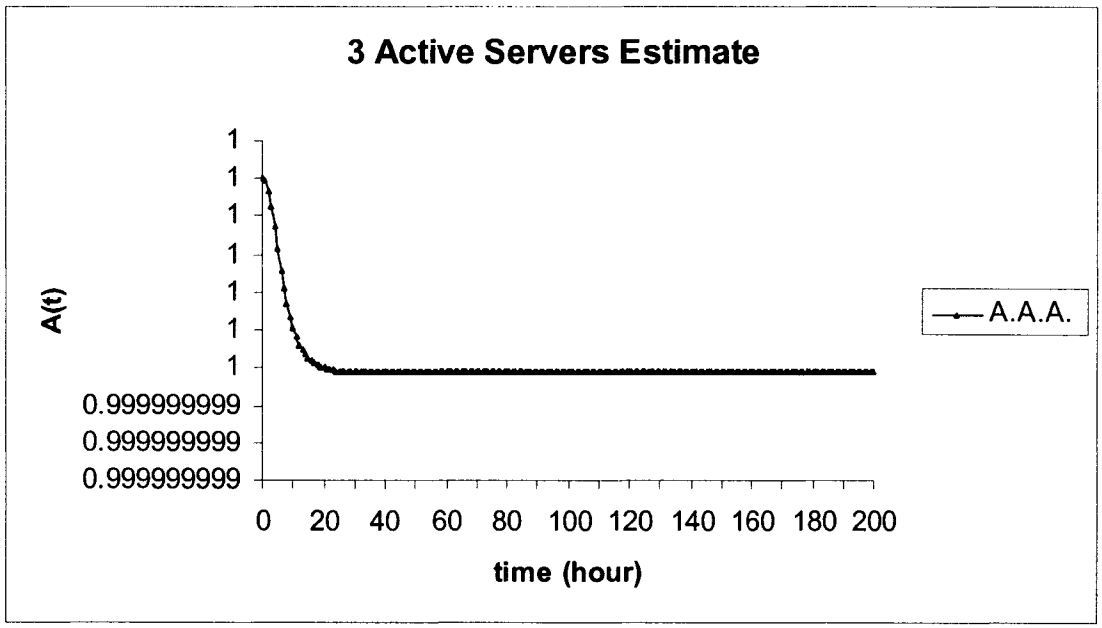

Figure 3.7 Three-Active servers availability estimate

CHAPTER 4

THE OBJECT-ORIENTED SPECIFICATIONS

4.1 Introduction

Markov processes, also known as continuous time Markov chains (CTMC) [50][51], have been widely applied to a variety of fields. A Markov Process $\{X(t), t \geq 0\}$ serves as a platform for modeling stochastic systems, particularly for evaluating reliability and performance of computing systems. However, complex systems often result in large Markov models, which are out of the logic view of the systems. Hence, it becomes a tedious task to insert all of the Markov states and transitions manually.

This dissertation proposes an object-oriented modeling framework for Markov chain specification and generation. The approach aims to facilitate computing systems' availability modeling. An availability model is specified in an XML [8] file, in which each component in the system is depicted as an object. Then the XML specification file is transformed into a corresponding Markov model, with a list of Markov states and a list of Markov transitions. After that, the Markov model is solved by a Markov chain evaluator.

4.2 Background and Related Work

Varieties of software packages exist to facilitate the reliability modeling and specify the models in compact forms [11][12][22][23][24][28]. UML is a widely-adopted modeling language used to visualize, specify, construct, and document the artifacts of a

software system [2]. UML not only encapsulates a rich set of diagrams, but also provide features such as stereotypes, tagged values, and constraints that can be customized and extended. Therefore, by adopting a subset of UML notations and formalizing them with proper semantics, it is feasible to transform the UML model automatically into corresponding analytical models.

Recently, researchers have attempted using UML to elaborate systems' dependability and performance aspects. ARAT [3][4] produces the dynamic metrics from UML usecase, statechart, and sequence diagrams for risk assessment at the architecture level. HIDE [5][6] supports dependability evaluation by elaboration of an automatic transformation from UML to Timed Petri Nets (TPN). Pai and Dugan [7] present an approach to automatically generate Dynamic Fault Trees (DFT) [28][42] from a UML system model. These approaches are similar in the way that UML diagrams are used to elaborate the systems, together with stereotypes, constraints, and tagged values; the difference is in embracing different diagrams, extensions, annotations, and in aiming at different systems.

Clearly, there is no unique solution to UML dependability modeling and its transformation. Indeed, UML is composed of a series of diagrams that depict the class structure, dynamic properties, and event sequencing for an object-oriented software system with no formal semantics attached to the individual diagrams. Therefore, it is impossible to apply rigorous automated analysis. However, formalizing a subset of UML diagrams to produce semantics to a given domain is feasible. UML formalization and transformation are still an ongoing research [8][9][10]

4.3 Overview

An object has a unique event based behavior in relevance to other objects in the system. We propose a scheme to describe the system's availability by adopting a subset of the object oriented features, and represent the objects in a XML format file. In this way, the XML representation of the system's reliability can be customized easily and configured during the runtime. The framework is depicted in Figure 4.1.

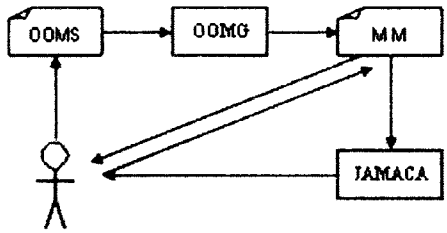

Figure 4.1 OOMSE Framework

OOMS represents the Markov chain specification in an object oriented fashion. The specification is a one-to-one mapping from the UML statechart diagrams. OOMG is the Markov chain generator, which transforms the OOMS into a list of the corresponding Markov states and a list of Markov transitions – a Markov Model (MM). The user can view and customize the Markov model. Then the Markov model is passed into the Java Markov chain analyzer (JAMACA) to be evaluated, and the result is returned to the user.

In the object-oriented availability specification paradigm, each component in the system is treated as an object. The system's availability model is actually delineated by the state changes of each object and the interactions among these objects. The corresponding Markov model is generated by all of the possible combinations of states in each object, together with the restrictions of guards, triggers and actions. Figure 4.2 gives a simple example of two objects' interaction.

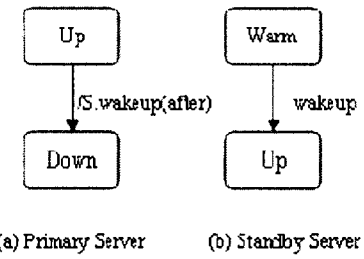

Figure 4.2 Statecharts for two servers

Figure 4.2 shows a computing system with primary server P and a standby server S in two statechart diagrams. Figure 4.2 (a) shows primary server and Figure 4.2 (b) shows the standby server. Initially, the primary server is working, and the standby server is in the warm state waiting to take over the control. The corresponding Markov state is specified as UW, where U denotes up state and W is for warm state. After t_1 time, the primary server fails, the transition from up to down is fired, and the transition in S labeled in wakeup is enabled, which change the state of S from warm to up in the time of t_2. The word "after" given in the parenthesis denotes that the action is taken after the transition. The sequence of Markov states is $UW \xrightarrow{t_1} DW \xrightarrow{t_2} DU$.

A subset of object-oriented structuring is adopted in the approach. We also extend the concepts by allowing a sequence of actions instead a single one. This can be achieved in the UML statechart diagram by separating individual actions by a special symbol, i.e., a comma. Moreover, an action is preceded with the respond object's name followed by the dot (.) operator. In this way, it speeds up searching for the right trigger. The modeling scheme is laid out and regulated in definitions given in the following section.

4.4 Definitions

Definition 4.1 An object is a 4-tuple $O = \{N, S, S_0, E\}$ where:

N – the unique name of the object

S – the set of possible states for the object

S_0 – the initial state for the object, and $S_0 \in S$

E – the set of events that can be generated by the object.

An event is defined as a timed event that changes the object from a source state to a destination state. Thus, an event is defined by $E = \{T\}$, where T is a set of state transitions for the event, $T = \{t_1, t_2, \cdots t_{|T|}\}$.

A transition consists of a source state, a destination state, a firing rate, and may optionally be associated with a trigger, a guard, and a sequence of actions. A transition is enabled either by a self generated timed event or triggered by another object's action. We assume all of the timed events and triggered events are exponentially distributed. A transition is regulated as follows.

Definition 4.2 A transition is a 6-tuple $t = \{s, d, r, tr, g, A\}$

where

s – source state of the transition

d – destination state of the transition

r – firing rate of the transition

tr – trigger of the transition

g – guard of the transition

A – a set of actions, $A = \{a_1, a_2, \cdots a_{|A|}\}$

4.5 Grammar

The grammar [68] for the XML representations is presented in this section, preceded with a list of acronyms for the grammar's readability.

Acronyms:

S – the start symbol
SU – system up
OBS – objects
OBN – object name
OB – object
OU – object up
STS – states
ST – state
TRANS – transitions
TRAN – transition
TG – trigger
TGN – trigger name
GDS – guards
GD – guard
ACTS – actions
ACT – action

Grammar:

S → <systemup>SU</systemup >
<objects>OBS</objects>
SU → (SU Op SU)|OU
OU → OBN=ST
OBS → OBS OB|OB
OB → <object name = String>
<states>STS</states>
<initial state>ST</state>
<events>TRANS</events>
</object>
TRANS → TRANS TRAN|TRAN
TRAN → <transition src=ST dst=ST rate=Num>
TG GDS ACTS </transition>
TG → <trigger>TGN</trigger>| ε
GD → <guard>GD Op GD|G </guard>| ε
G → <guard>OBN [= =|!=]ST</guard>| ε
ACTS → ACTS|ACT
ACT → <action>OBN.TGN(before| after)</action>| ε
OBN → String

$STS \rightarrow STS|ST$
$ST \rightarrow String$
$TGN \rightarrow String$
$Op \rightarrow$ [&&| ||]
$String \rightarrow$ [a-zA-Z0-9]$^{+}$
$Num \rightarrow$ [0-9]$^{+}$.[0-9]$^{+}$

Terminals = <, >, /, systemup, objects, object, name, states, state, initial events, transition, src, dst, rate, trigger, guard, action, before, after.

4.6 Algorithm

In this section, we briefly illustrate the algorithm that transforms a list of objects into CTMC. For the sake of clarity, the algorithm is broken into five major procedures, namely the main procedure, generate states, process transitions, perform transition, and perform action. Pseudo – Java code is used to depict the algorithm for convenience. Members of objects are accessed by the dot operator (.), i.e., t.guard; meanwhile, members of lists and strings are accessed via the [ ] operator, as they are in arrays, i.e., state[i].

4.6.1 The Main Procedure

The main procedure takes in a list of objects. It maintains also three lists, namely a list of old states, a list of new generated states, and a list of generated transitions, as global variables. The procedure first creates the initial Markov states via synthesis all of the objects' initial states, separated by commas, and adds the initial state to the new state list. Then it takes a state from the head of the new list, calls the generate procedure (which generates new states and transitions) to handle the state, appends the state to the end of the old list, and remove the state from the new list. Finally, it marks the "good" states in the old list. The following gives a skeleton of the procedure.

While the new list is not empty

Do

state = head of the new list

call generate (state);

append state to old list;

remove state from new list;

4.6.2 Generate States and Transitions

The generate state procedure takes a state (a string) as a parameter, and generates new states and new transitions whenever it is possible. The procedure traverses the object list and the transitions in each object, and then it calls the process transition procedure to handle each transition.

4.6.3 Process Transitions

The process transition procedure takes a global state, an object state, the object position in the object list, and a transition of the object as parameters. The procedure first checks whether the transition meets the three conditions: (1) there is no trigger (a transition with a trigger cannot be fired by itself), (2) the guard is satisfied, and (3) the transition's source meets the current object state. Consequently, the procedure checks whether there is any actions associated with the transition. If there is no action, the transition is fired accordingly. If the action is characterized as "before", as indicated by the action's parameter, the action is fired before firing the transition. Conversely, the transition is fired before the action, provided that the action is marked as "after".

4.6.4 Perform Transition

The perform transition procedure first creates a new state by replacing the current state's *i*th component with the destination of the transition (state[i] = t.dst). Secondly, it creates a new Markov state transition by setting the transition's source to the old state, and destination to the new state, together with the transition rate. Furthermore, the procedure adds the new state to the new state list, and the new transition to the transition list.

4.6.5 Perform Action

The perform action procedure first finds the object position via the object name specified in the action, then locates the transition that possesses the trigger, and finally calls the perform transition procedure to fire the transition, provided that the guard condition is satisfied and the transition's source meets the current object state.

4.7 Examples

This section gives two examples to illustrate the methodology. The first example consists of two objects, while the second example has three objects interacting with each other.

4.7.1 Example 1

We adopt the HA-OSCAR system [47] as a target model, with minor modifications for the sake of simplicity. The system has a primary server P and a warm-standby server S. The primary server provides the services and processes all the user's requests. The standby server is waiting to take over the control when a failure happens in the primary server. When the primary server fails, after a certain time, the monitoring facility will detect this failure, and the standby server will be "waken up" and takes over

the control. Once the primary server gets repaired, it will take back the control of the system, and put the standby server back to "dormant." Table 4.1 gives the XML specification for the two servers.

Table 4.1 OOMS of two servers in XML

```
<systemup> (P=U || S=U) <systemup>

<objects>
<object name="P">
      <states>
          <state name="U"/>
          <state name="D"/>
      </states>
      <initial state="U"/>
  <events>
      <transition src="U" dst="D" rate="tp1"/>
         <action>S.wakeup(after)</action>
      </transition>
      <transition src="D" dst="U" rate="tp2">
       <action>S.dormant(before)</action>
      </transition>
   </events>
</object>

<object name="S">
   <states>
          <state name="W"/>
          <state name="U"/>
          <state name="D"/>
   </states>
   <initial state="W"/>
   <events>
       <transition src="W" dst="U" rate="ts1">
         <trigger>wakeup</trigger>
       </transition>
       <transition src="W" dst="D" rate="ts2"/>
       <transition src="U" dst="D" rate="ts3"/>
       <transition src="U" dst="W" rate="ts4">
         <trigger>dormant</trigger>
       </transition>
        <transition src="D" dst="W" rate="ts5">
          <guard>P==U</guard>
        </transition>
      </events>
  </object>
<objects>
```

In Table 4.1, the first line specifies that the system requires either P or S to be functioning. The primary server consists of two states, up (U) and down (D), while the

standby server has an additional warm (W) state. Initially, the primary server is functioning, and the standby server is in the warm state. Hence, the initial Markov state is UW. When the primary server fails after a certain time *tp*1, it goes to state D, and the standby server is brought to state U by the trigger "wakeup" after time *ts*1. The action wakeup takes a parameter "after" to indicate the action needs to be performed after the transition is fired. The sequence of generated Markov states is UW→DW→DU. After it gets repaired, the primary server will go to state U again, and leave the standby server to state W. The generated Markov states are DU→DW→UD. The action dormant takes a parameter "before" to indicate the action needs to be performed before the transition fired. The standby server can fail when it is in both warm and up states. It can be repaired only when the primary server has not failed, and this is guarded by the condition P==U. Table 4.2 and Table 4.3 list the generated Markov states and transitions.

Table 4.2 Markov states for two servers

#	States	Up
1	U,W	Y
2	D,W	
3	D,U	Y
4	U,D	
5	D,D	Y

Table 4.3 Markov transitions for two servers

#	Source	Destination	Rate
1	U,W	D,W	tp1
2	D,W	D,U	ts1
3	U,W	U,D	ts2
4	D,W	U,W	tp2
5	D,W	D,D	ts2
6	D,U	D,W	ts4
7	D,U	D,D	ts3
8	U,D	D,D	tp1
9	D,D	U,D	tp2

4.7.2 Example 2

We use a hypothetical system as the second example, by adding an additional "cold" standby server `C` to the previous example. Initially, `C` is in the "cold" state. It will be brought to the "warm" state whenever the primary server or the warm standby server fails. It will be in the "up" state when both the primary server and the standby server failed. It will be put back to "sleep" when the first two servers are healthy. The cold server can fail in both the "up" and the "warm" state, but no failure will happen when it is in the "cold" state.

The difference from the previous example in specification is that when the primary server fails or get repaired, it needs to enable the triggers in the "warm" standby server and the "cold" standby server, sequentially. While the "warm" standby and the "cold" standby servers need to specify guard conditions to prevent unnecessary events to happen. For example, if the Markov state is `UDW`, and a failure happens to the first server, the first state goes from `U` to `D`, and then changes the Markov state to `DDW`. The corresponding actions try to wake up the second or the third server by enabling the triggers. However, the second and the third servers are in state `D` and `W`, therefore, the `S.wakeup` and `C.warmup` triggers cannot enable the transitions and are ignored. Nevertheless, the `C.wakeup` trigger changes the Markov states from `DDW` to `DDU`. The complete specification of the three servers is given in Table 4.4, and the generated Markov states and Markov transitions are given in Table 4.5 and Table 4.6.

Table 4.4 XML specification of three servers

```
<systemup> (P=U || S=U||C=U) <systemup>

<objects>
<object name="P">
     <states>
          <state name="U"/>
          <state name="D"/>
     </states>
     <initial state="U"/>
  <events>
     <transition src="U" dst="D" rate="tp1"/>
        <action>S.wakeup(after)</action>
        <action>C.warmup(after)</action>
        <action>C.wakeup(after)</action>
     </transition>
     <transition src="D" dst="U" rate="tp2">
      <action>S.dormant(before)</action>
        <action>C.dormant(before) </action>
        <action>C.sleep(before) </action>
     </transition>
   </events>
</object>

<object name="S">
   <states>
         <state name="W"/>
         <state name="U"/>
         <state name="D"/>
   </states>
   <initial state="W"/>
   <events>
<transition src="W" dst="U" rate="ts1">
          <guard>P==U</guard>
         <trigger>wakeup</trigger>
       </transition>
       <transition src="W" dst="D" rate="ts2">
           <action>C.warmup(after) </action>
           <action>C.wakeup(after) </action>
       </transition>
       <transition src="U" dst="D" rate="ts3"/>
           <action>C.warmup(after) </action>
           <action>C.wakeup(after) </action>
       <transition src="U" dst="W" rate="ts4">
         <trigger>dormant</trigger>
       </transition>
        <transition src="D" dst="W" rate="ts5">
          <guard>P==U</guard>
           <action>C.sleep(before) </action>
        </transition>
     </events>
</object>
```

Table 4.4 Continued

```
  <object name="C">
    <states>
<state name="C"/>
          <state name="W"/>
          <state name="U"/>
          <state name="D"/>
    </states>
    <initial state="W"/>
    <events>
        <transition src="C" dst="W" rate="tc1">
            <guard>P==D || S==D</guard>
            <trigger>warmup</trigger>
        <transition src="W" dst="D" rate="tc2"/>
        <transition src="W" dst="U" rate="tc3">
            <guard>P!=U && S!=U</guard>
            <trigger>wakeup</trigger>
            <action>C.warmup(after) </action>
        </transition>
        <transition src="U" dst="D" rate="tc4"/>
        <transition src="U" dst="W" rate="tc5">
            <guard> S!=D</guard>
            <trigger>dormant</trigger>
        </transition>
        <transition src="W" dst="C" rate="tc6">
            <guard>P!=D && S!=D</guard>
            <trigger>sleep</trigger>
         </transition>
         <transition src="D" dst="C" rate="tc7">
            <guard>P!=D && S!=D</guard>
          </transition>
      </events>
   </object>
<objects>
```

Table 4.5 Markov states for three servers

#	States	Up
1	U,W,C	Y
2	D,W,C	
3	D,U,C	Y
4	D,U,W	Y
5	U,D,C	Y
6	U,D,W	Y
7	D,D,C	
8	D,D,W	
9	D,D,U	Y
10	D,W,W	
11	D,U,D	Y
12	U,D,D	Y
13	D,D,D	
14	D,W,D	
15	U,W,D	Y

Table 4.6 Markov transitions for three servers

#	Source	Destination	Rate
1	U,W,C	D,W,C	tp1
2	D,W,C	D,U,C	ts1
3	D,U,C	D,U,W	tc1
4	U,W,C	U,D,C	ts2
5	U,D,C	U,D,W	tc1
6	D,W,C	U,W,C	tp2
7	D,W,C	D,D,C	ts2
8	D,D,C	D,D,W	tc1
9	D,D,W	D,D,U	tc3
10	D,U,C	D,W,C	ts4
11	D,U,C	D,D,C	ts3
12	D,U,W	D,W,W	ts4
13	D,W,W	D,W,C	tc6
14	D,U,W	D,D,W	ts3
15	D,U,W	D,U,D	tc2
16	U,D,C	U,W,C	ts5
17	U,D,W	D,D,W	tp1
18	U,D,W	U,D,C	tc6
19	U,D,W	U,D,D	tc2
20	D,D,C	U,D,C	tp2
21	D,D,W	U,D,W	tp2
22	D,D,W	D,D,D	tc2
23	D,D,U	D,D,W	tc5
24	D,D,U	D,D,D	tc4
25	D,W,W	D,D,W	ts2
26	D,W,W	D,W,D	tc2
27	D,U,D	D,W,D	ts4
28	D,W,D	U,W,D	tp2
29	D,U,D	D,D,D	ts3
30	U,D,D	D,D,D	tp1
31	U,W,D	D,W,D	tp1
32	U,W,D	U,D,D	ts2
33	U,W,D	U,W,C	tc7

4.8 UML Availability Modeling

This section illustrates using UML to specify cluster systems' availability. In the model, all of components are treated as objects, and transitions are treated as object interactions. A subset of UML notations is adopted, namely class diagram and statechart diagrams, to specify cluster systems' availability. HA-OSCAR is used as an example to illustrate the concept.

The system model is described using a class diagram with the keyword "system" as its name to indicate it is the entry point of the model. The system model is divided into (composite) submodels, in a "tree like" structure, associated with the aggregation relation, which indicates the "has a" relation, between a submodel and its higher model. A submodel can have submodels or "leaves". The "leaves" are object entities defined either by a statechart diagram or a predefined formula. Failure rate and repair rate are defined as tag-value pairs. "Siblings" relations are denoted by associations, which are stereotyped as "series" or "parallel" optionally with a predefined formula as its name.

The servers are modeling by using state chart diagrams. A statechart diagram consists of states and transitions, and has the following properties:

- The state from the UML initial state is represented as a filled black circle to indicate the system's initial state.
- A state has a name and a tag to indicate the component's state status, namely, Good: true or false.
- A transition is associated with two tags: failure rate and repair rate.
- A transition may consist of guards and actions.
- Actions are denoted further by regular expressions.

Figure 4.3 shows the Availability Model of HA-OSCAR as an example. The system is divided into a client submodel and a server submodel, with the "series" relation. The client submodel consists a clientnodes model, which is denoted as the k-out-of-n formula in the method field of the class diagram. The server submodel "has" the PServer and the SServer submodels, with the "parallel" and the "CTMC" relation. Then two server submodels are needed to be defined in statechart diagrams.

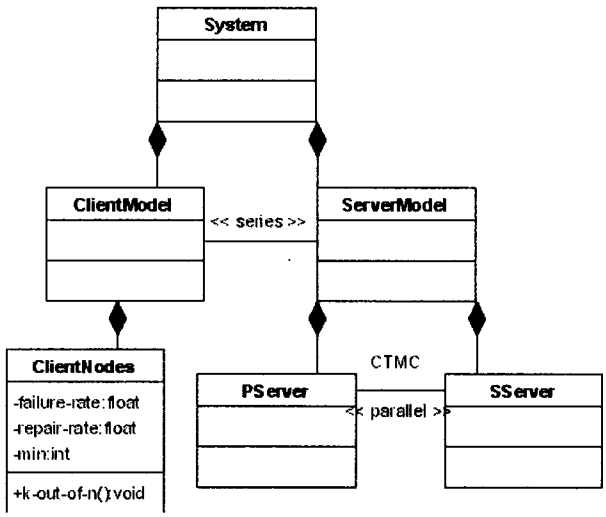

Figure 4.3 Availability model of HA-OSCAR

Figure 4.4 shows the primary server in a statechart diagram. Initially, the primary server is in the up state. After a period of time, it can fail; thus, it goes to fail state. The monitor will detect this event and bring the primary to down state and "wake up" the standby server. The "wake up" transition can only happen if the standby server is in the warm state (w). This is denoted by the "guard" followed by the action, which will change the standby server to the up state. Once the primary server is in the down state, it undergoes the repairing process. After it gets repaired, the system will reconfigure, put the standby server into the "dormant" state, and bring the primary server to the up state. The transition to "dormant" can only happen if the standby server is in the up state.

Figure 4.5 shows the statechart diagram for the standby server. Initially, the standby server is in the "warm" state (W), waiting to take over control of the system. The transition from W to U and from U to W are denoted as guard "prohibit" to indicate the transition cannot happen by itself. The transition from down state (D) to warm state (W) is also guarded since the primary server has the priority to get repaired.

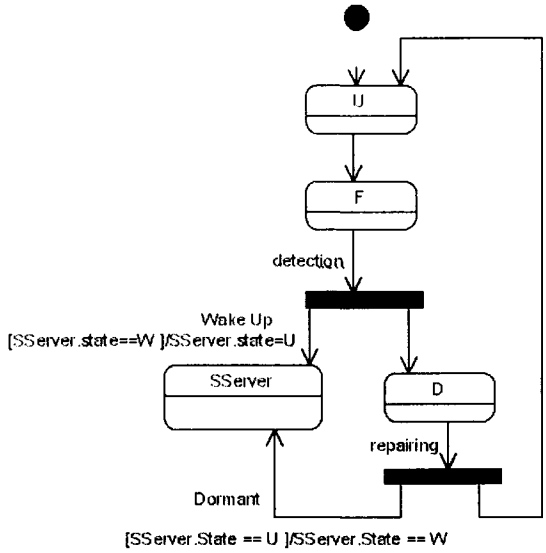

Figure 4.4 Statechart diagram for the primary server

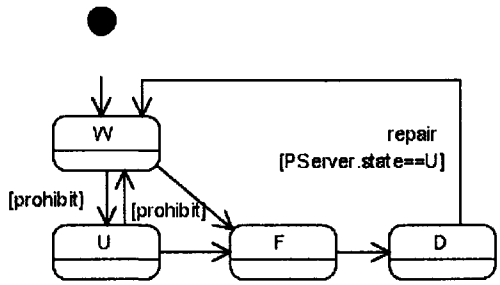

Figure 4.5 Statechart diagram for the standby server

CHAPTER 5

NUMERICAL SOLUTIONS OF MARKOV PROCESSES

5.1 Introduction

Markov processes, also known as continuous time Markov chains (CTMC), have been widely applied to a variety of fields. A Markov Process serves as a platform for modeling stochastic systems, particularly for evaluating dependability and performance of computing systems. When the transition matrix of a Markov process is large, it becomes very difficult to obtain a closed form solution for the transient state probability. In this case, a numerical approach is normally the choice. There are fruitful numerical methods for this purpose [53][60]. Among these, the uniformization (or randomization) method attracts more interests for its series computing and bounded error control properties.

In this chapter, we first review some existing numerical methods, formalize and simplify the uniformization procedure, then discuss several alternative implementations and pitfalls of the procedure, and finally propose a light-weight model for solving large sparse Markov processes.

5.2 Numerical Methods Overview

The numerical solutions of the Markov models can be classified into two categories: methods for solving the steady state and methods for the transient solution. This section gives an overview of the numerical methods in these two categories.

5.2.1 Steady State Solutions

The steady state solution of a Markov model may or may not exist. It depends on the structure of the matrix Q. If the steady state solution of a Markov model exists, it follows that the rate change of the probability vector $\pi(t)$ at the steady state is zero, i.e., $\frac{d\pi(t)}{dt} = 0$. Therefore, for a homogeneous CTMC, from Equation 2.18, the following equation holds [31][50][51][54]

$$\pi Q = 0 \tag{5.1}$$

where Q is the infinitesimal generator matrix.

For a discrete time Markov chain (DTMC), let P be the probability transition matrix, then from Equation 2.17, the steady state solution can be written as

$$\pi P = \pi \tag{5.2}$$

which is equivalent to

$$\pi(I - P) = 0 \tag{5.3}$$

The stationary probability distribution vector π also satisfies

$$\pi e = 1 \tag{5.4}$$

as shown in (2.20), where e is a vector with all elements are 1s, i.e., $e = \{1, 1, \cdots, 1\}$.

The Equation 5.1 and 5.3 can be solved by the direct methods, iterative methods [53][54]. Direct methods include Gaussian elimination, LU decomposition, etc. Iterative

Methods include Jacobi method, Gauss-Seidel method, SOR method, and a family of projection methods.

5.2.2 Transient Solutions

The transient solution of a Markov model is to find the probability vector $\pi(t)$ at time t, by solving the Chapman-Kolmogorov differential equation $\frac{d\pi(t)}{dt} = \pi(t)Q$. The explicit solution is in the form of $\pi(t) = \pi(0)e^{Qt}$. There are many different ways to solve the above equation [50][51][52][54][69], such as matrix decomposition, matrix scaling and powering, uniformization, ordinary differential equation (ODE) solvers, and Krylov subspace method. Currently, most of the existing software packages include multiple methods inside, and the most suitable method can be chosen either by the modeler [23] or automatically detected by the software itself [55].

5.3 The Uniformization Procedure

A continuous time Markov chain (CTMC) $\{X(t), t \geq 0\}$ on a finite state space, with the probability transition matrix P and the infinitesimal generator matrix Q, can be described by the forward Kolmogorov differential equation [50][54]:

$$\frac{dP(t)}{dt} = P(t)Q \tag{5.5}$$

Let $\pi(0)$ and $\pi(t)$ be the initial probability vector and the probability vector of the CTMC at time t, respectively ($\pi(t) = \pi(0)P(t)$), then Equation 5.5 can be expressed as

$$\frac{d\pi(t)}{dt} = \pi(t)Q \tag{5.6}$$

and the solution to Equation 5.6 is [54]

$$\pi(t) = \pi(0)e^{Qt} \tag{5.7}$$

The matrix exponential e^{Qt} is defined by the Taylor series as

$$e^{Qt} = \sum_{i=0}^{\infty} \frac{(Qt)^i}{i!}$$

Let $\Gamma \geq \max_i |q_{ii}|$, where q_{ii} denotes the diagonal element of the infinitesimal generator matrix Q, and P be a discretized stochastic probability matrix, such that $P = I + \frac{1}{\Gamma}Q$. Together with Equations 5.5 and 5.6, the transient probability vector at time t can be obtained by computing the following formula:

$$\pi(t) = \pi(0)\sum_{i=0}^{\infty} P^i e^{-\Gamma t} \frac{(\Gamma t)^i}{i!} \tag{5.8}$$

The above procedure that discretized the continuous time Markov chain is known as the uniformization method [61][62][63]. The transient solution is computed from the discrete time Markov chain, which is embedded in a Poisson process with rate Γt. The uniformization procedure establishes the equivalency between continuous and discrete time Markov chain.

5.4 Implementation Analysis

5.4.1 Truncation Error

In implementation, the infinite series of Equation 5.8 needs to be truncated at a certain point to meet the desired error tolerance. The error bound formula is given by [54] [57]:

$$1-\sum_{i=0}^{k} e^{-\Gamma t}\frac{(\Gamma t)^i}{i!} \le \varepsilon \tag{5.9}$$

where ε is the error tolerance, and k is the truncation point of that is large enough to satisfy the desired error control ε.

Therefore, the actual formula (5.8) is reduced to compute the finite series as follows:

$$\pi(t)=\pi(0)\sum_{i=0}^{k} P^i e^{-\Gamma t}\frac{(\Gamma t)^i}{i!} \tag{5.10}$$

As shown in [57][58][59], choosing the error tolerance ε between 10^{-11} and 10^{-12} will achieve the maximum accuracy.

The truncation point k is referred to as the "right truncation point" for the Poisson process [59]. Some researchers [60][61][62] suggest also "left truncation" on the series, as given in the following formula

$$\pi(t)=\pi(0)\sum_{i=l}^{k} P^i e^{-\Gamma t}\frac{(\Gamma t)^i}{i!} \tag{5.11}$$

where l is the left truncation point.

The reason for performing the left truncation is that the Poisson distribution becomes thin as Γt grows, and the terms on the left side are small and less significant, therefore, they can be "omitted". Reibman [57] gives a formula for calculating the left bound in the appendix of the paper. However, this technique is not adopted in our solution for the following reasons:

a) As Γt is small, so will the left truncation point; then there will be no need for the left truncation. As indicated in [57], for $\varepsilon = 10^{10}$, left truncation is not used if $\Gamma t < 25$.

b) There is no better way to compute the initial matrix power P^l than by vector matrix multiplication. As a result, the truncation simply ignores the $e^{-\Gamma t}(\Gamma t)^i / i!$ factor. Therefore, there is no advantage in doing so.

5.4.2 General Implementation

With a given error tolerance, the right side truncation point k of the infinite series in Equation 5.8 is computed. The procedure is naturally implemented by vector matrix multiplication instead of computing the matrix power, for the sake of reducing computation complexity. This is done by simply moving the initial probability vector $\pi(0)$ inside the summation, in which case formula 5.10 becomes

$$\pi(t) = \sum_{i=0}^{k} \pi(0) P^i e^{-\Gamma t} \frac{(\Gamma t)^i}{i!} \tag{5.12}$$

The rest of the procedure is to compute the vector matrix multiplication iteratively and add the result to the previous one.

The term $e^{-\Gamma t}$, referred to as "scalar", may be applied to π after the iteration [54][59] rather than applied to $\pi(0)$ at the initialization step. Nevertheless, when Γt is large, the term $(\Gamma t)^i$ may grow fast and potentially cause an overflow problem. As a consequence, the computation either encounters an exception or increases result inaccuracy, depending on the platform the program is executing. Introducing $e^{-\Gamma t}$ to the procedure at the very beginning will keep the intermediate results growing slower, although the results of the beginning iterations steps are small. However, it will not be troublesome if Γt is small, as for example in partitioning the time line into multiple time intervals.

5.4.3 Multiple Time Intervals

When the time t is large, difficulty arises as $e^{-\Gamma\Delta t}$ is too small and $(\Gamma t)^i$ is too large, causing the computation to underflow and overflow. For such a case, it is necessary to divide the time into multiple time intervals [54], then calculate the probability vector at each time interval, and use the results as the initial value for the next iteration.

For example, the total time $(0,t)$ is partitioned into $l+1$ steps $t_0, t_1, \ldots t_l = t$ with equal length Δt, and $t_{i+1} = t_i + \Delta t$, then the probability vector at time t_{i+1} is computed as follows:

$$\pi(t_{i+1}) = \sum_{j=0}^{k} \pi(t_i) P^j e^{-\Gamma\Delta t} \frac{(\Gamma\Delta t)^j}{j!} \tag{5.13}$$

The error control used for computing the truncation point k should also be adjusted to reflect the fact. Stewart [54] suggests using ε/l as the error control value for each time interval, so that the overall error is bounded by the desired error tolerance ε.

5.4.4 Small State Spaces

For a system with a small state space, i.e., a system with several hundred states, the infinitesimal matrix can be implemented in a two dimensional array. With the time partitioned into equal time intervals, the right truncation point k should be the same for each time interval. The term $T = \sum_{j=0}^{k} P^j e^{-\Gamma\Delta t} (\Gamma\Delta t)^j / j!$ is a common factor and can be stored in memory, thus it can be used in all the time steps [54][63]. The probability vector at time t_{i+1} is simply computed by $\pi(t_{i+1}) = \pi(t_i) T$. We give an analysis on the complexity of this method as follows.

Calculating the matrix power P^j for each term is unrealistic. A better way to solve T is to use the iteration method as shown in Figure 5.1.

Let $T = e^{-\Gamma \Delta t}$

for $j = 1$ to k do

$$T = T + TP\frac{(\Gamma \Delta t)}{j}$$

Figure 5.1 Algorithm to compute the T term

It will requires $O(k \cdot n^3)$ multiplications to solve T and $O(n^2)$ multiplications for $\pi(t_i)T$. Hence, it requires $O(l \cdot n^2 + kn^3)$ multiplications to reach the final solution at t. If the method is implemented as described in Section 5.3, it will need $O(lkn^2)$ multiplications. To find out if this approach is efficient or not, we compare the operation cost with the vector-matrix multiplication approach. The storing T approach is more efficient than then the vector-matrix multiplication if it satisfy the following

$$l \cdot n^2 + kn^3 < lkn^2 \tag{5.14}$$

By canceling the common factor n^2, we get

$$l + kn < lk \tag{5.15}$$

and finally we have

$$l > \frac{k}{k-1} n \approx n \tag{5.16}$$

Therefore, we conclude that for a system with a small state space, and the infinitesimal generator is implemented in a two dimensional array, storing the common

factor matrix T for computing the probability vector is more efficient if the divided time interval l is larger than the system state space.

5.4.5 Stiffness Models

Stiffness problem often arises in dependability and performance models, causing numerical instability in solution methods. A Markov model is said to be stiff when $\max_i |\mathrm{Re}(\lambda_i)| >> \min_i |\mathrm{Re}(\lambda_i)|$, where λ_i are the eigenvalues of the infinitesimal generator matrix Q [54]. A stiff Markov process often has one or more states with greatly different rates. To solve the problem, Reibman and Trivedi [57] suggest choosing small time-scale as a function of the slow rate. The experiment in [57][58][59] shows that the calculation is more expensive for stiff Markov models.

5.5 Large Sparse Matrix

For system availability and performance models, the infinitesimal generator matrix Q of the corresponding Markov chain is generally large with thousands of states where most elements are zeros [55]. This is especially true when the Markov model is generated automatically from a higher level description, i.e. Stochastic Petri Net [22]. In the M/M/1 [31][52] queuing system, for example, the corresponding infinitesimal generator matrix Q contains three elements in each row (with two elements off the diagonal) and the rest are zeroes. For a large sparse Markov model, the two-dimensional array approach is infeasible because of the limited storage and enormous computing operations. Sparse storage and preserving methods are used to solve the storage problem and computational complexity, normally by utilizing sparse matrix implementation techniques. There are numerous software packages, such as Linpack [65], JMP [66] for this purpose. Jin and Ziavras [64] present a parallel programming technique for large

sparse matrix multiplication. These packages and techniques are normally equipped with full matrix operational features. Hence, utilizing these packages will inevitably increase the computing code size, resulting too complex to be deployed and distributed. Furthermore, these general purpose matrix packages may not be efficient for such specific computing process. A better way is to tailor these packages to suit the uniformization method.

SERT [63] stores the state space, events, and transition rates into vectors, and traverse these vectors when computing Markov models. It does solve the large and sparse problem. The shortcoming of this approach is its lack of modularity, from the point of view of software development and maintenance. It smears the interface between the algorithm and the matrix computation. This makes it difficult to implement the algorithm and to upgrade the software package thereafter.

We propose our Light Weight Markov Chain solution (LWMC). Our goal is to create a LWMC calculator, which can be used to handle large sparse Markov models. In our approach, the transition matrix consists of n column vectors, where n is the number of states of the Markov model. Only nonzero elements are stored in each column vector together with their position. The approach is based on the following observation.

Let c_j denote the j th column vector of the discretized probability generator matrix P, and $(\pi P)_j$ is the j th component of the vector-matrix multiplication πP, then $(\pi P)_j = \pi c_j$. When performing the multiplication, only nonzero components of c_j are used, indicated by the indexes of the nonzero components of c_j. The operation can be expressed as $(\pi P)_j = \sum_i c_{ij}\pi_i$, for all i and $c_{ij} \neq 0$. Our solution is to implement

Equation 5.13 and a column-wised sparse matrix equipped with a vector-matrix multiplication operation only. In this way, the algorithm and the matrix implementations are separated, leading to a convenient development and maintenance of the software package.

CHAPTER 6

MONITORING AND ANALYSIS

6.1 Introduction

In this chapter, we discuss a reliability-aware monitoring and modeling framework which provides near real-time system availability/reliability analysis and information for high performance cluster computing systems. Our work aims to address issues in existing solutions in which HPC system management only considers performance aspects and leaves reliability to a reactive (i.e. addressing issues after they happen) or manual recovery approach. Our proposed framework dynamically obtains availability information such as failure and repair events of the individual nodes and is able to model and evaluate system availability for the overall and partial HPC system. With near-real-time availability evaluation, the framework enables runtime systems such as schedulers or resource managers to be aware of more accurate system reliability and hence better utilization and efficiency of the HPC systems. Lastly, we demonstrate usefulness of our approach to a scheduling runtime system based on the availability information provided by this framework. The failure and analysis model was reconstructed from system logs of the Lawrence Livermore National Laboratory Advanced Simulation and Computing (ASC) machines. The data set was used to

understand system availability and validate how a scheduler can exploit such information to improve the overall completion time for parallel jobs in the presence of failures.

Certainly, the framework can be extended to include more features such as more detailed error classification, failure correlation, and distributions other than the exponential distribution with additional efforts.

6.2 General Terms and Concepts

In this section, we give a brief description on the general terminologies and measuring concepts.

6.2.1 Fault, Error, and Failure

Definition 6.1 A fault is an anomalous physical phenomena, either internal caused by manufacturing problem, fatigue, design flaw or external disturbance, such as environmental perturbations, temperature, vibration, etc.

Faults can be classified into transient faults, intermittent faults, and permanent faults. A transient fault is a fault resulting from temporary environmental conditions. An intermittent fault is a fault that is only occasionally and unexpectedly present due to unstable hardware or software states. A permanent fault is a fault that is continuous, persistent and stable due to an irreversible change.

Definition 6.2 An error is an undesired system behavior, when the system is not able to deliver services complying with what is expected of system. An error is a manifestation of a fault.

Definition 6.3 A failure is the occurrence of an undesired circumstance affecting the service of the system. The system is unable to perform some action that is due or expected. It is caused by an effective error that affects the delivered service.

In general, an error is caused by a fault, and a failure is caused by an effective error. Figure 6.1 shows the relationship of the three.

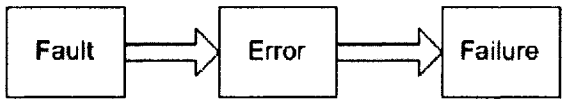

Figure 6.1 Fault, failure and error

Since we are interested in system availability analysis, a failure of a component is considered to be an event that causes the component out of service such that the affected component cannot response to any request. A failure of a system means the outage of the system.

6.2.2 General Concepts

The sample space of an experiment, is the set of all possible outcomes of that experiment [70][71][72]. Let $X_1, X_2, \cdots, X_n$ be the random variables form a random sample of size n from some distribution, the sample mean $\overline{X}$ is given by

$$\overline{X} = \frac{1}{n}\sum_{i=1}^{n} X_i \tag{6.1}$$

and the sample variance is given by:

$$S = \frac{1}{n-1}\sum_{i=1}^{n}\left(X_i - \overline{X}\right)^2 = \frac{1}{n-1}\left[\sum_{i=1}^{n} X_i^{\,2} - \frac{1}{n}\left(\sum_{i=1}^{n} X_i\right)^2\right] \tag{6.2}$$

It has been proved that $\overline{X}$ is the best estimator of the mean and S is the best estimator of the variance [71].

Goodness-of-Fit Test. Let N_i $i = 1,2,\cdots,k$ be the number of observations in the random sample, with the sample size $N = \sum_{i=1}^{k} N_i$, and let E_i be the expected value of

type i. If the null hypothesis H_0 is true, and the sample size is large, then the distribution of the statistic $Q = \sum_{i=1}^{k}\left[\frac{(N_i - E_i)^2}{E_i}\right]$ will be approximately a χ^2 distribution with $k-1$ degree of freedom. It is desired to carry out the test at the significance level of α_0. The null hypothesis H_0 should be rejected if Q is in the $1-\alpha_0$ quantile of the χ^2 distribution with $k-1$ degree of freedom. The test is called the χ^2 test of goodness-of-fit test [72].

If the sample mean $\overline{X}$ is used to calculate the statistic Q, then the approximate distribution of Q when H_0 is true lies between a χ^2 distribution with $k-2$ degree of freedom, and this leads to the following formula:

$$\chi^2 = \sum_{i=1}^{k}\left[\frac{(N_i - E_i)^2}{E_i}\right] = \chi^2_{k-2,1-\alpha_0} \tag{6.3}$$

As an example, consider a component in which the failure time is distributed with probability density function $f(t)$. The observation starts from t_0 to t_k with k time intervals, $\Delta t_1 = t_1 - t_0$, $\Delta t_2 = t_2 - t_1$, $\Delta t_k = t_k - t_{k-1}$. And the number of failures observed in each interval $N_1, N_2, \cdots, N_k$ accordingly is given in Table 6.1 below.

Table 6.1 Example 1

Time intervals	# failures
Δt_1	N_1
Δt_2	N_2
...	...
Δt_k	N_k

The expected number of failure E_i during interval Δt_i is

$$E_i = N \int_{t_{i-1}}^{t_i} f(t)dt \tag{6.4}$$

6.3 Related Work

Availability and reliability are key attributes of computer systems. Event logs that are generated by the monitoring facilities provide an effective means of identifying defects to improve availability, and they are used in a variety of ways. The monitoring mechanisms were originally developed to meet the needs of hardware designers who wanted to debug their products. These mechanisms have been developed later on to meet other needs. Hence, the system logs are the result of an evolutionary process rather than being the result of a predefined plan. Event logs have been used in many ways, including long term trend analysis, online diagnosis use for failure prediction, and MTTF estimation. The log analysis process is highly dependent upon the quality and completeness of the event logs. If the information is incomplete or missing, it will be difficult or impossible to interpret the events' activities.

There is a wide variety of research that is based upon the analysis of event logs. Lin et al. [73] analyze the error log file on file servers to demonstrate the log is composed of at least transient and intermittent processes. Wein and Sathaye [74] present their experience with validation of complex computer system availability models. Ram et al [75] measures the failure rate in widely distributed software. Chillarge et al. [76] presented a failure rate measurement technique on distributed software, based upon classifying failure data into "failure windows." Moran et al. [77] illustrated the availability monitoring facility developed at Digital Equipment International. These

approaches are similar in performing data analyses; the difference is the way they classify errors, the correlation, distribution, and aims at different models.

The uniqueness of our approach is that our framework automatically performs data analysis and availability modeling and repository when there are failure/repair events detected to provide near-real time availability information and inventory of the HPC system. Existing approaches either perform the analysis manually or retrieve the data from the database (logs) periodically in order to generate the analysis report. We envision that reliability-aware runtime system can exploit near-real time availability information to improve efficiency and better HPC resource utilization.

6.4 Overview of the Framework

The monitoring framework consists of two major parts, namely reliability-ware monitoring and system availability modeling and analysis. The system availability modeling module provides a near-real-time availability evaluation for both node-wise and overall system. Currently, we constructed a proof-of-concept for each individual module. However, we plan to integrate our framework with the availability and system configuration and build an availability inventory and configuration database with normalization capability for the actual node-wise and system's mean time to fail (MTTF) and mean time to repair (MTTR). Figure 6.2 shows the reliability-aware monitoring and modeling framework.

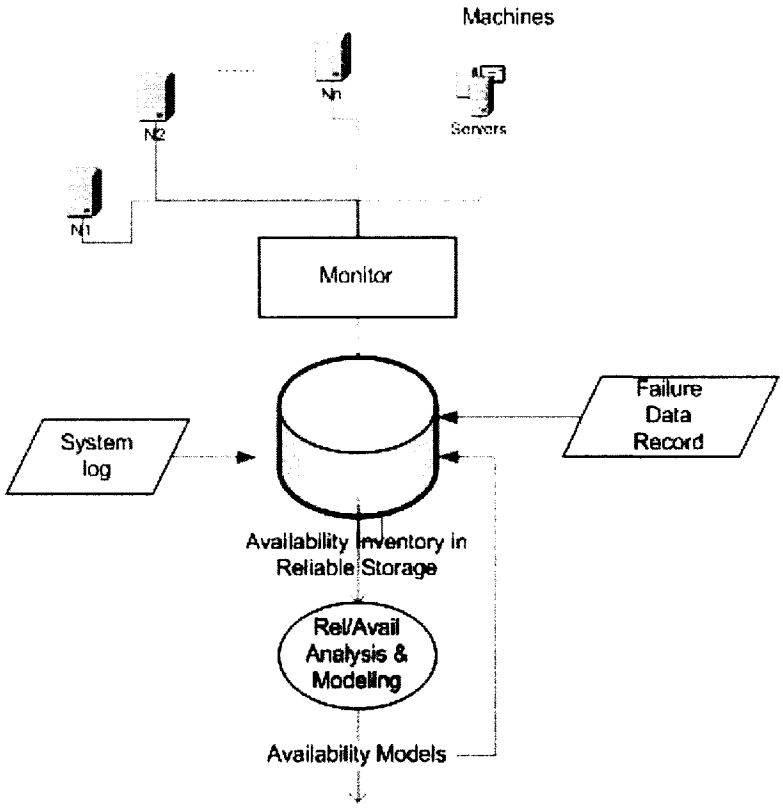

Figure 6.2 Monitoring and analysis framework

In this framework, the monitoring facility is responsible for detecting failures, repairing other types of events, and recording these events into the system log. The failure data record (FDR) is stripped from the system log file and contains only the events that are necessary to evaluate the system's availability/reliability. The availability models are used to evaluate the system's availability, which are stored in an XML file. The system's log, failure data record, and the availability model are stored in a reliable disk storage. The Analysis & Solution module is responsible for pulling the data from the FDR, doing the analysis, and feeding the result into the availability models. The framework consists of three functionalities: (1) detection, which is responsible for detecting failure events based on the failure classification, (2) logging, which writes the failure events into the system log file and the system failure data record, and (3) analysis

and update, which is responsible for failure analysis and normalization of the current system's availability. Figure 6.3 shows the flow diagram inside the monitoring and analysis framework.

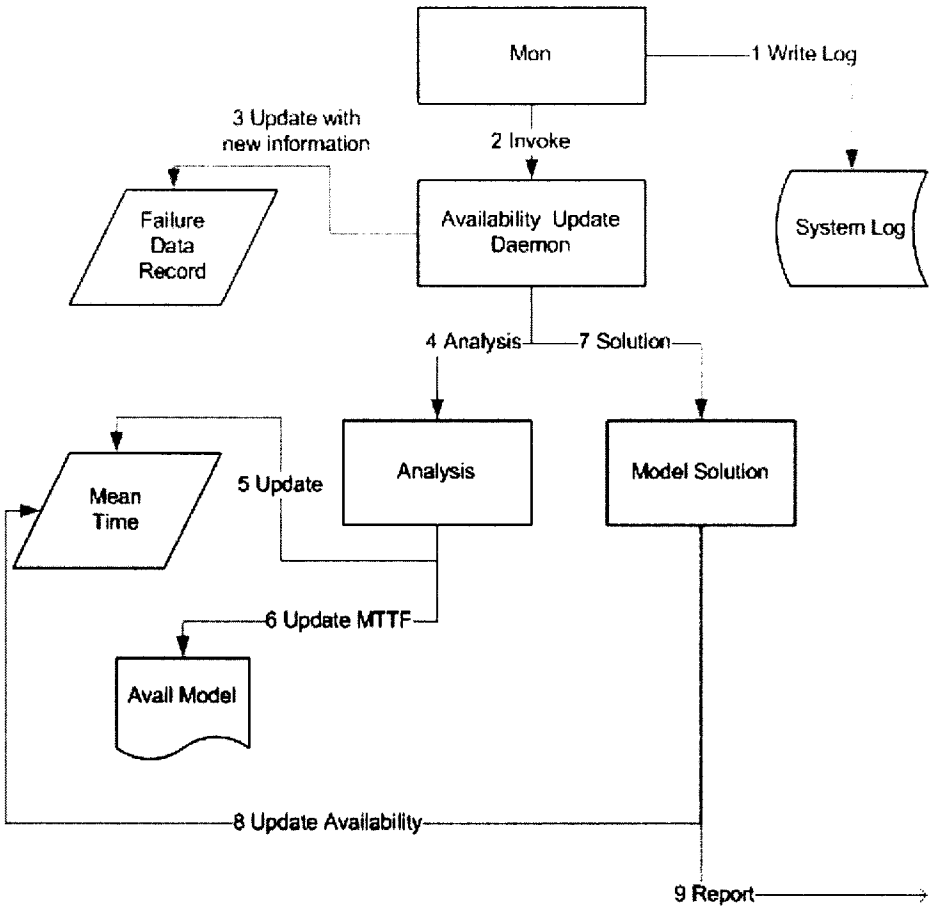

Figure 6.3 Monitoring and analysis flow diagram

In Figure 6.3, each arc in this diagram is associated with a number to indicate the flow sequence, and a name to denote the action. The monitoring facility (MON) is responsible to watch the system health. Once a failure or repair event is detected, MON writes this information into the system log, and invokes the availability update daemon (AvailUpd) to update the system's failure data record (FDR). After that, the AvailUpd invokes the analysis module, which queries the FDR to get the recent

failure/repair activity, reevaluates the MTTF, MTTR, etc., and then updates the mean time (MT) and the specification of the availability model with this new information. Finally, the AvailUpd invokes the solution module to solve this availability model. The result of availability solution is written back to availability repository on reliable storage.

As mentioned earlier, the monitoring system will maintain the configuration file in the system database to describe the availability property for each component. The file is a XML output from our design and availability analysis framework. Figure 6.4 shows a snippet for a server and a node instances in the availability and mean time file (MT). Each instance in the MT file has seven fields: (1) the starting time of the instance t_0, (2) the current time t_1, (3) the total elapsing time T in hours, which equals to $t_1 - t_0$, (3) total number of failures TF during this period of time, (4) the total downtime TDT, which represents the total repair time, (5) the MTTF, which equals to T/TF, (6) the MTTR, which equals to TDT/TF, and (7) the steady state availability of the instance, which can be acquired from Equation 2.12. Among these fields, only t_1 and TF are recorded for each failure, and TDT is recorded for each repair. The rest of the fields are updated based on the changes of t_1, TF and TDT.

Once the AvailUpd finishes updating the MT file, it will pass these information to the solution engine to have the availability result. Figure 6.5 and Figure 6.6 show the system and the server availability model. The AvailUpd daemon first evaluates the servers' availability; it updates the MTTF and MTTR in the servers' availability model with the newly updated information in MT, and then passes the servers' availability mode to the Markov solution engine to have the servers' availability.

```
<SystemAvailability>0.999928</SystemAvailability>

<Servers>
    <Server 1>
        <t0>7/21/2000</t0>
        <t1>10/1/2004</t1>
        <T>27960</T>
        <TF>9</TF>
        <TDT>1392.5</TDT>
        <MTTF>3106.27/MTTF>
        <MTTR>154.72</MTTR>
        <Availability>0.952</Availability>
    </Server 1>

        .....
</Servers>

<Nodes>
        .....
    </Node435>
        <t0>7/21/2000</t0>
        <t1>10/1/2004</t1>
        <T>27960</T>
        <TF>33</TF>
        <TDT>1157</TDT>
        <MTTF>847.27/MTTF>
        <MTTR>35.06</MTTR>
        <Availability>0.952</Availability>
    </Node435>

    .....

</Nodes>
```

Figure 6.4 MT file for a single instance

Once the servers' availability is solved, the `AvailUpd` evaluates the nodes' availability. It first takes the mean of the MTTF and MTTR, invokes the k-out-of-n computing facility by passing in the number of computing nodes needed, and the total number of available nodes. The system's availability is the product of the servers' and the nodes' availability.

```
<System>

    <Servers>
        <Availability>CTMC(serverObjs.xml)</Availability>
    </Servers>

    <Nodes>
        <Availability>K-OUT-OF-N(K,N) </Availability>

    </Nodes>

</System>
```

Figure 6.5 System availability model

```
<Objects>
<Object Name="A1">
        <states>
                <state name="U"/>
                <state name="D"/>
        </states>
        <Good>
                <state name="U"/>
        </Good>
        <Status state="U"/>


        <Events>
                <Transition src="U" dst="D" rate="0.000321"/>
                <Transition src="D" dst="U" rate="0.006463"/>
        </Events>
</Object>


<Object Name="A2">
        <states>
                <state name="U"/>
                <state name="D"/>
        </states>
        <Good>
                <state name="U"/>
        </Good>
        <Status state="U"/>


        <Events>
                <Transition src="U" dst="D" rate="0.000357"/>
                <Transition src="D" dst="U" rate="0.00766"/>
        </Events>
</Object>

</Objects>
```

Figure 6.6 Servers availability model

The `AvailUpd` daemon first evaluates the node-wise and system-wise availability and then updates the availability slots in the MT file. Once the availability is calculated, the `AvailUpd` evaluates the availability of the nodes and the entire system's

availability. The update facility performs the data analysis, and updates the MT file. The failure and repair events are assumed to be exponentially distributed, and computes the mean, variance and does the goodness-of-fit test. Finally, it generates a report and updates the system's availability. Figure 6.7 shows the flow diagram for the data analysis module.

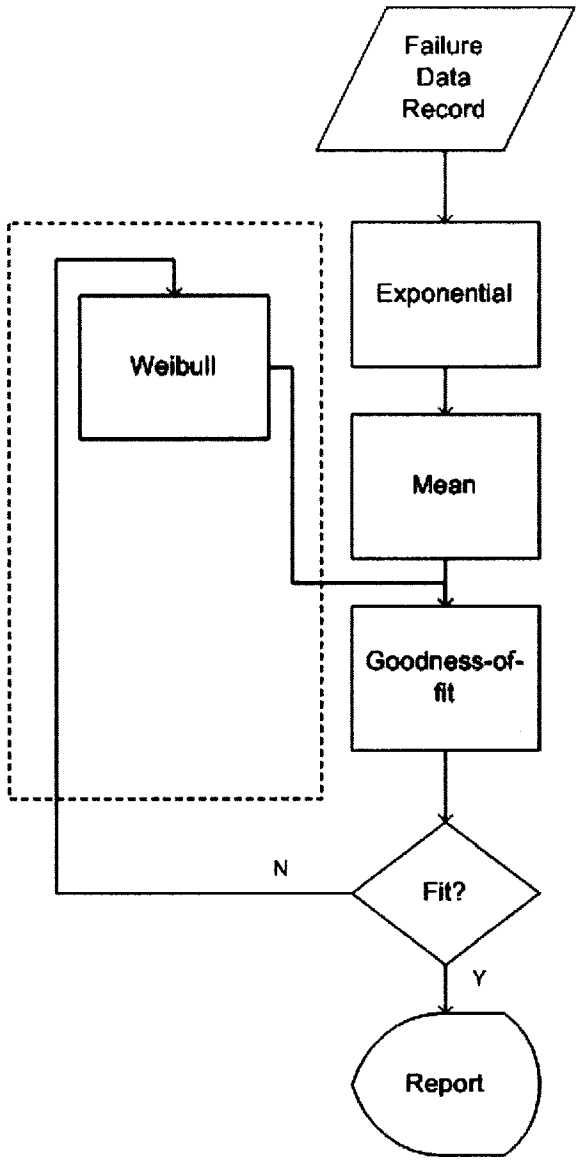

Figure 6.7 Data analysis flow diagram

6.5 Measuring and Analysis

We analyzed the system logs of major HPC computing infrastructure provided from Lawrence Livermore National Laboratory. The system log file contains significant system events, from years past, collected from four ASC machines, namely White, Frost, Ice, and Snow. We then performed a detailed analysis on these data sets. For the purpose of brevity, we present only the analysis result of White. White, the largest among the aforementioned systems, is a 512-node, 16-way symmetric multiprocessor (SMP) parallel computer. All nodes are of IBM's RS/6000 POWER3 symmetric multiprocessor 64-bit architecture. Each node is a stand-alone machine possessing its own memory, operating system (IBX AIX), local disk, and 16 CPUs. Table 6.2 lists a sample of events in an ASC White machine during the four year period, from July 21, 2000 to October 1, 2004.

Table 6.2 Example of failures in White

Id	Type	Subtype	Wk-endng	TDT (hr)	Sect	Host list
1914	HW	HW-SSA_ADAPTER	7/28/2000	2	whit	265
1917	HW	HW-IO	7/28/2000	33	whit	275
1931	HW	HW-SWITCH	7/28/2000	72	whit	275
1913	HW	HW-SSA_ADAPTER	7/28/2000	2	whit	287
1968	HW	HW-SWITCH	8/4/2000	137	whit	017
1952	HW	HW-CPU	8/4/2000	19	whit	025
1938	HW	HW-CPU	8/4/2000	23	whit	026
1953	HW	HW-MEMORY	8/4/2000	21	whit	067
1954	HW	HW-MEMORY	8/4/2000	20	whit	100
1949	HW	HW-MEMORY	8/4/2000	21	whit	266
1986	HW	HW-CPU	8/11/2000	74	whit	010
1983	HW	HW-MOTHERBOARD	8/11/2000	76	whit	026
1969	HW	HW-OTHER	8/11/2000	48	whit	032
1970	HW	HW-CPU	8/11/2000	21	whit	052
1971	HW	HW-MOTHERBOARD	8/11/2000	20	whit	113
1985	HW	HW-IO	8/11/2000	74	whit	115
1980	SW	SW-COMM_SS	8/11/2000	172	whit	128
1972	HW	HW-CPU	8/11/2000	46	whit	194
1973	HW	HW-MEMORY	8/11/2000	48	whit	211
1974	HW	HW-MEMORY	8/11/2000	144	whit	241
2005	HW	HW-CPU	8/18/2000	144	whit	019
2002	HW	HW-IO	8/18/2000	188	whit	026

Table 6.2 shows the failure id, the type and subtype of the failure, date of the failure discovered, total down time (TDT) cost by this event, and the system affected by this event. Note that the TDT is the repair time for this failure, which includes the response time, resolution time, and the verification time. We analyzed the availability, MTTF and MTTR for each node in the system. The MTTF for the a node equals to `(total elapsed time)/(number of failures)`. The average MTTF for each node in the system is approximately 3923.8 hours.

The MTTR is the `(total down time)/(number of failures)`, which implies that it approximately needs this much time to recover from each failure event. The average MTTR for each node in the system is approximately 55.3 hours. The steady state availability for each node is 0.98. Figure 6.8 shows the availability for each node in the White cluster system.

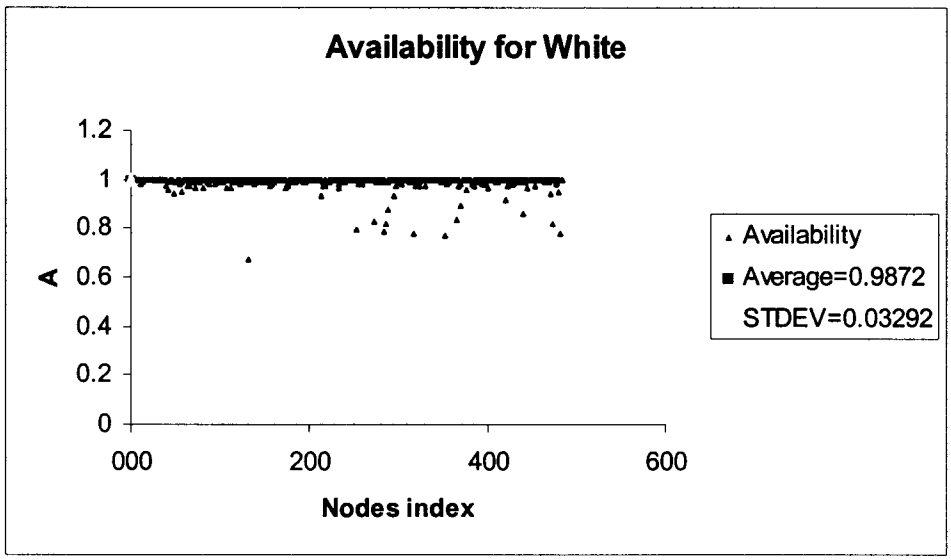

Figure 6.8 Availability of each node in the White system

From Figure 6.8, we can see that the majority of the availability of each node is above 0.95, and a few of them are below 0.8. The reason could be some nodes had been

used extensively compared to the others. For the login nodes (assume they are the servers), the average MTTF is 1997.5 hours, and the average MTTR is 112.3 hours.

Figure 6.9 and Figure 6.10 show the mean time to fail (MTTF) and total down time (TDT) for each node in the cluster White, the means are 3293 and 355 hours, and the standard deviation is 1217 and 56, respectively. From Figure 6.9, we observe that the MTTF for each node varies, namely, the smallest MTTF is 230 hours, and the maximum is 5592 hours. In Figure 6.10, node downtime density indicates that the most of the total downtime for each node are around 100 hours; some failure events cost more time to be fixed, thus increasing the total average TDT.

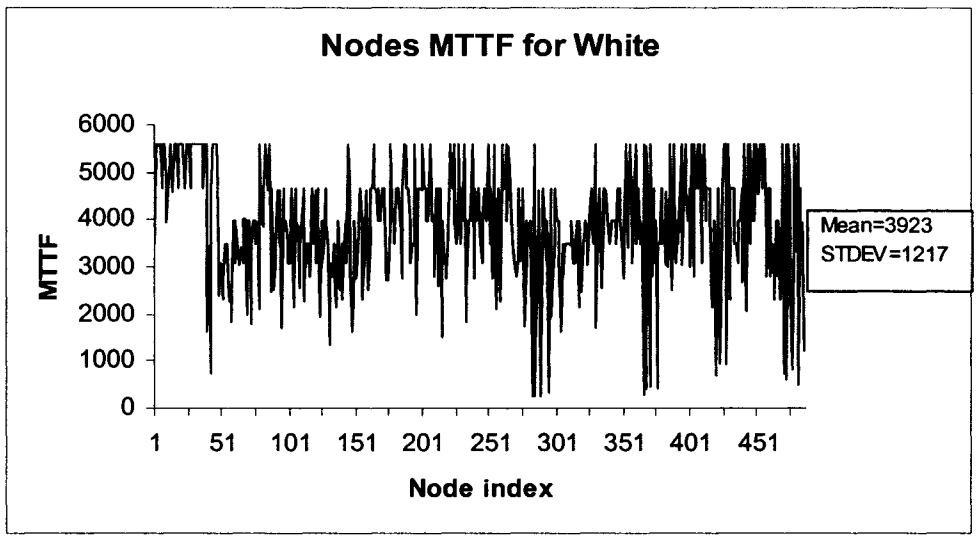

Figure 6.9 Nodes MTTF density

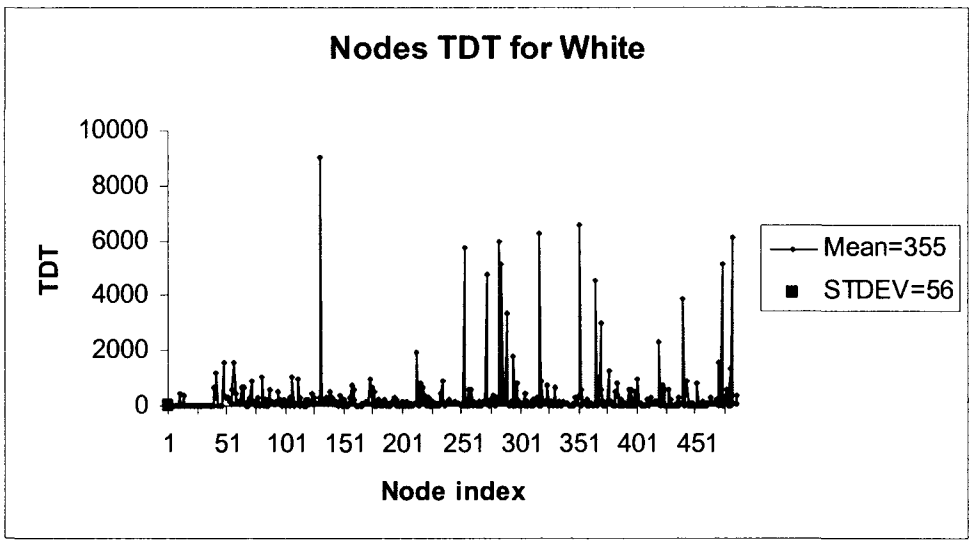

Figure 6.10 Node downtime (in hours)

6.6 Improvement Analysis and Comparison

In this section, we give an example to illustrate the improvement of our approach to provide more up-to-date system availability information.

The monitoring framework presented here updates the availability information whenever there is a failure and repair event occurs. Other existing approaches evaluate these information during a period of time, such as once a month. The information collected reflects the availability aspects of the system during this period, and is lagging from the actual system's behaviors. On the other hand, our monitoring framework presents more accurate results comparrd with others. The dynamic information can help dependability-aware scheduling and check-pointing to perform their tasks more efficiently. We will use the events in node 012 of White to illustrate the concepts.

Table 6.3 shows the events happened in node 012 of cluster White. We apply both our technique and a typical approach (assume periodically evaluation on the first day of

each month) with these events. Initially (July 21, 2000), we obtain the MTTF, MTTR and Availability for node 012 is 1997, 112, and 0.98, respectively.

Table 6.3 Events in node 012 of White

Failures	Id	Wk-ending	TDT(hr)	Node Id
1	2062	9/8/2000	22	012
2	2609	12/22/2000	36	012
3	2724	1/19/2001	12	012
4	3032	3/9/2001	97	012
5	3085	3/16/2001	2	012
6	3299	5/4/2001	22	012
7	3440	6/1/2001	57	012
8	4023	9/7/2001	40	012
9	4060	9/21/2001	37	012
10	4487	1/18/2002	88	012

During the first event happened on September 8, 2000, our approach is able to capture this episode immediately, and availability information was updated with MTTF = 1176, MTTR = 22, and availability = 0.9812. The monthly update technique would still have the old information (MTTF is 1997, MTTR is 112 and availability is 0.98) until October 1, 2000. From September 8, 2000 to October 1, 2000, the information about node 012 is obsolete, because it does not reflect the event happened on September 8, 2000. The situation is getting worse if there are more events happening during the update period, since the availability information is affected more than a single event.

Figure 6.11 shows the MTTF changes for node 012, both for dynamic and monthly updates. We can see that there is a lagging for the monthly update. There are a few points that are different from the monthly update. It is because there are more than one events happening during that period, and the monthly update techniques can not reflect this episode. Similarly, Figure 6.12 and Figure 6.13 show the MTTR and availability changes on node 012 for both ours approach (referred as dynamic) and monthly update techniques.

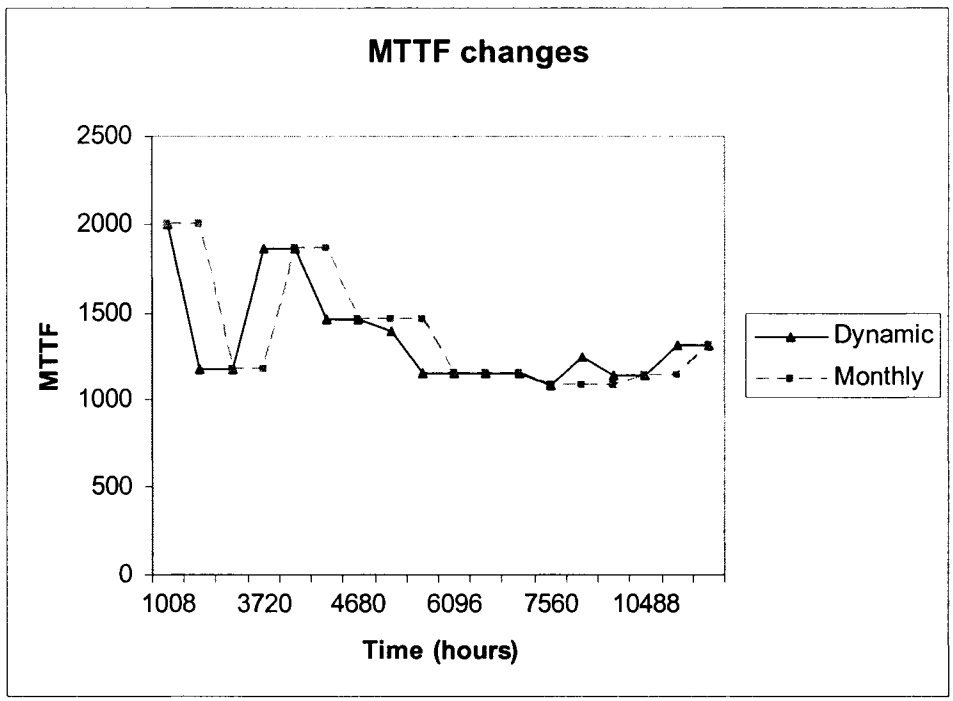

Figure 6.11 MTTF changes of dynamic and monthly updates

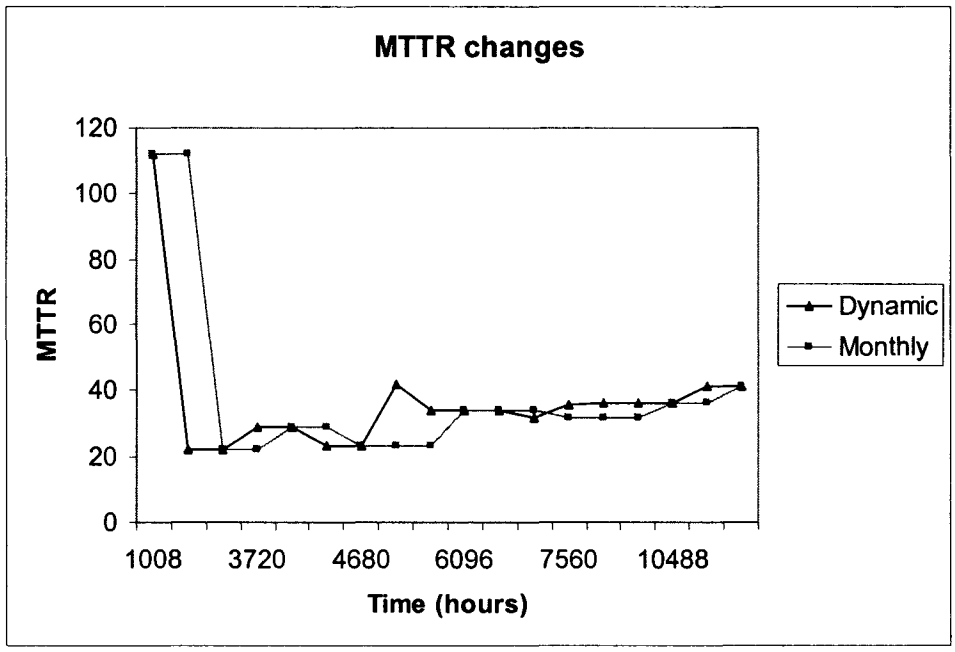

Figure 6.12 MTTR changes of dynamic and monthly updates

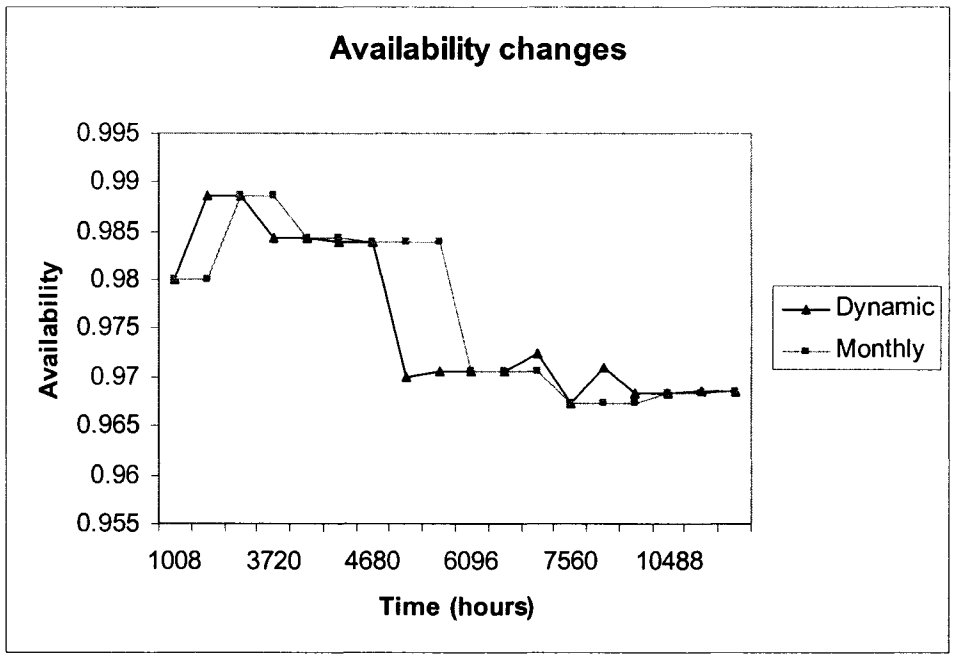

Figure 6.13 Availability changes of dynamic and monthly updates

6.7 Reliability and Availability Aware Scheduling

The monitoring framework provides up to date availability and reliability information for the overall system and also individual nodes. With this information, job scheduling can be improved effectively. In this section, we present an experiment to demonstrate the affect of considering reliability and availability parameters in the scheduling algorithm.

We have used the events of cluster White to develop scheduling algorithms that use the availability information to schedule parallel jobs. The parameters MTTF, MTTR, and the elapsed time obtained from the information service are dynamically updated through a monitoring system. There are various important parameters that are significant in developing an effective scheduling algorithm such as job completion time, performance, throughput, utilization, reliability, safety, queuing times, etc. Here, we

consider reliability as an important attribute for a scheduling algorithm and show how the job completion time is affected when choosing such an algorithm, since the reliability information can be easily acquired from the MTTF provided from the monitoring system.

Figure 6.14 illustrates a Gantt chart [78] which shows the effectiveness of completion time for MPI jobs, in the presence of node failures. As the number of nodes increases, the probability that one of the nodes fails also increases, thus affecting the overall job completion time. In the case of parallel programs, the failure will affect the job running on all the machines. The MTTF for n nodes is given by the following equation:

$$MTTF(n,\lambda) = \frac{1}{n\lambda} \tag{6.5}$$

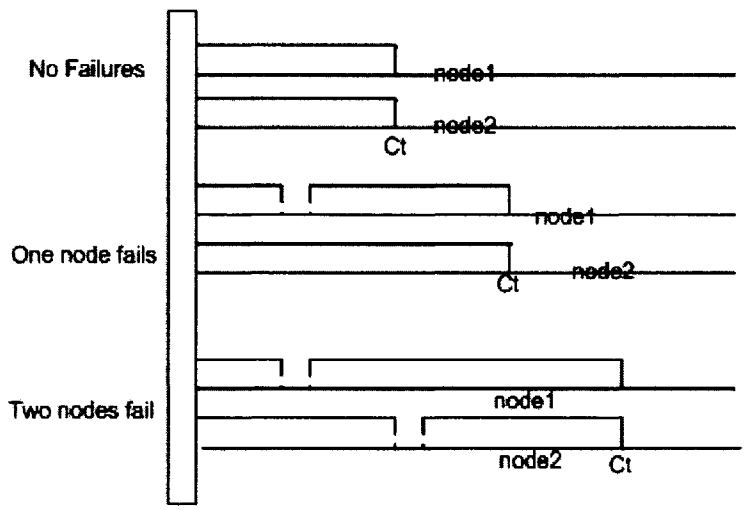

Figure 6.14 Completion time for a parallel job impacted by node failure

Availability (or reliability) of the computing nodes becomes a very important factor in scheduling parallel jobs (such as MPI), because the job must be restarted and/or reallocated to a different set of nodes when failures occur. In this case, the completion time of the job will be affected in the event of failure, as shown in the Gantt chart above.

According to Amdahl's Law, the maximum speedup achievable is limited by a serial fraction of the program. The "speedup" of a parallel program is defined to be the ratio of the rate at which job is run on N processors to the rate at which it is executed on one node. The speedup *S(N)* is given by [78]:

$$S(n) = \frac{1}{\frac{p}{n} + (1-p)} \tag{6.6}$$

where p is the fraction of code that can be made parallel (therefore, $1\text{-}p$ is the code that has to be executed sequentially) and n is the number of nodes. The expected execution time on n nodes $T(n)$ is given by:

$$T(n) = \frac{T(1)}{S(n)} \tag{6.7}$$

A job completion time can be described as:

$$Ct = T(n) + T(f) \tag{6.8}$$

Where $T(n)$ is the expected completion time and $T(f)$ is the total time spent on the nodes that have failed to run the job, p the fraction of code that is made parallel is assumed to be 0.891 hereafter.

Node-wise MTTF and the number of cooperating nodes affect the total system availability and reliability as well as a completion time. For a given node-wise failure rate, the total system MTTF decreases (i.e. the frequency of failure increases) as the number of nodes on which the job runs increases. At some points, scalability will approach a break-even point where a long running job will not be able to finish due to its completion time being longer than the total system MTTF.

Figure 6.15 shows how the MTTF varies with the increase in the number of nodes for three different failure rates. This phenomenon also elicits a conclusion that runtime systems, such as a scheduler aware of system reliability (MTTF), will benefit from MTTF information, especially more accurate MTTF information from our near real-time modeling approach. Since there are situations where a job will never complete due to a relatively short system MTTF, our future work will consider reliability-aware check-pointing technique [80][86] that aims to derive an optimal interval to save application contexts based on the runtime system MTTF. In addition, we will investigate a check-point and restart time as one of the repair events, and its time factor that will influent our total system availability model.

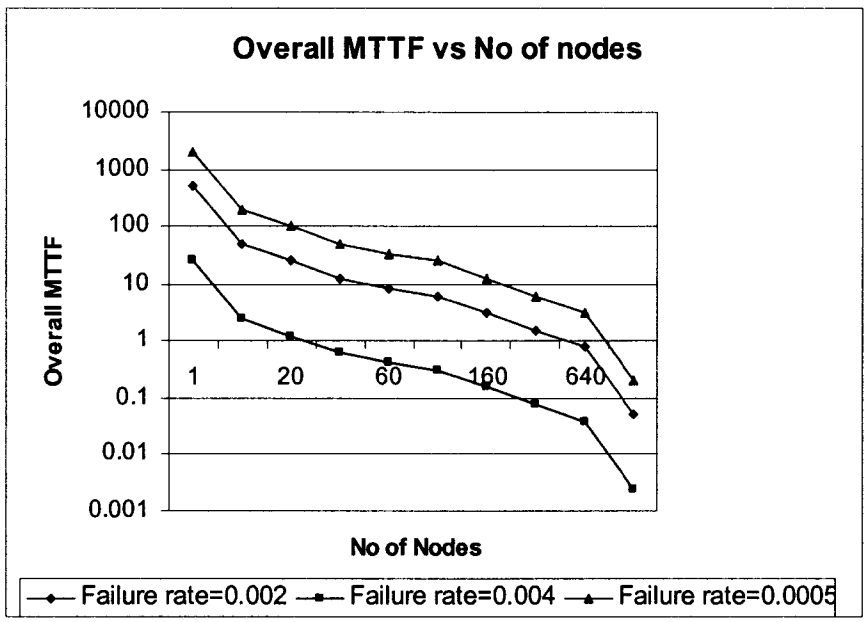

Figure 6.15 MTTF on various numbers of nodes

CHAPTER 7

CONCLUSION AND FUTURE WORK

This dissertation presents a novel technique to facilitate the availability modeling, runtime monitoring, and near real time availability evaluation. The background and basic concepts of modeling techniques were reviewed first. We then characterized HPC system modeling as our targeted problem domain. Three key components are described in the dissertation, namely availability modeling, model evaluation, and monitoring and analysis. The HPC cluster system's availability model is divided into submodels based on their functionalities, and these submodels are either represented by series structures or Markov models. An object-oriented Markov model specification has been developed to facilitate availability modeling and runtime configuration. We reviewed some numerical solution methods for solving Markov models. We also suggested a light weight solution for solving large sparse Markov models. The method has certain advantages for its modularity, platform independency, and small size.

In the monitoring and analysis chapter, we presented a framework to enable automatically data analysis and availability update. This framework not only is an important stepping stone to enable runtime systems to be aware of resource availability, but also ensures the more accurate result with dynamic analysis approach, hence making better decisions in unleashing HPC power. We analyzed the actual data based on a real-

world production system logs from the Lawrence Livermore ASC machines, and fed analysis result to validate our approach. We have also demonstrated that an impact to runtime system performability is due to system reliability/availability information.

The key contribution of this dissertation is that we developed a new specification technique for cluster availability modeling. The specification is more intuitive, comparing with other existing solutions. The technique can reduce the burden from the modeler's shoulder. As the result, the availability model specification can be updated easily to enable runtime availability evaluation. The technique provides a mile stone for UML availability modeling, which is a one-to-one mapping from the intermediate model specification.

Currently, we consider each computer as a single instance under the assumption such that each instance having an exponential distribution with failure rate λ, that means aging has no significant effect. The theory of Markov processes assumes that the waiting time in a state before a transition to another state occurs, is a random variable having an exponential distribution. The future work should extend the model to include aging. This can be done by considering semi-Markov model [50], where the failure is not exponential but may be Weibull, or Gamma distributions. The modeling framework can also be extended to capture more detailed system behaviors, such as software failure. The monitoring and analysis facility needs to be able to diagnosis the software defects as well, and detailed failure classification. Furthermore, modeling evaluation should include more methods, and be able to choose the appropriate method(s) for a particular model. The system's availability model is under the assumption such that, if the monitoring cannot receive the response of any node, it considers that the node to has failed. This deficiency

can be extended by monitoring and modeling more detailed events and instances in each node and probing critical services or applications of interest. Non-homogeneous Markov model and semi-Markov model [71] should also be investigated to represent the system's behaviors.

APPENDIX A

OBJECT-ORIENTED MARKOV MODEL TRANSFORMATION

Algorithm: Generate state-spaces and transitions

Note:

- The algorithm is written in pseudo – Java code
- Members of an object are accessed using the dot operator, i.e., t.guard
- Members of lists, strings are accessed via the [ ] operator, as they are in arrays, i.e., state[i]

Main procedure

Given: a list of objects

Output: a list of Markov states with "good" states are marked, and a list of transitions

Variables:

state – a global state (object states separated by commas)
ol – an old list
nl – a new_list
objl – an list of objects
transitions – a list of transitions

Procedure:

```
nl = initialStates(objl);
while (nl != empty){
        state = nl[0];
        generate (state);
        ol.append  (state);
        nl.remove (state);
}

mark_goods( ol );
output (ol);
```

Procedure generate(state)

Given:

state – a global state

Global variables:

objl – an list of objects

Local variables:

t – a transition
tl – a list of transitions
s – an object state
o – an object
i, j – iteration counter

Procedure:

```
i=j=0;
s = state[i];

while(i<objl.size()){

        o = objl[i];
    tl = o.transitions;

        for(j = 0; j<tl.size(); j++){
      t = tl[j];
   processTransition (s, state, t, i);
        }

        i ++;
        s = state[i];
}
```

Procedure processTransition (s, state, t, i)

Given:

s – an object current state
state – a global state
t – an transition
i – the current object position

Local variables:

src – a source state

Procedure:

```
If (t.trigger != null) return; //do nothing

src = t.src;
if (src != s) return;
```

```
if(!satisfyGuard(state, t.guard)) return;

if ( t.action == null ){
      performTransition (state, t, i);
}else{
     if (isActionBefore (t.action)){
        performAction (state, t.action);
        performTransition (state, t, i);
      }else{
        performTransition (state, t, i);
        performAction (state, t.action);
      }
}
```

Procedure performTransition (state, t, i)

Given

state – a global state
t – an transition
i – the current object position

Global variables

tl – a list of transitions

Local variables

ns – a new state

Procedure

```
ns = replace(state, i, t.dst); //state[i] = t.dst
if(!transitionExists(s, n, tl)){
   tl.add(new Transition(s, n, t.rate));
}
addNewState(n);
```

Procedure performAction (state, action)

Given

state – a global state
action – an action

Variable

i – an integer, indicate an object position
t – a transition

Procedure

```
//1. find the object position
i = indexOfObjectByAction (action);

//2. traverse the transitions in the object, find //the correct trigger
t = findTranByAction (i, action);

//3. check if the t.src = = state[i]
 if( t.src != state[i] ) return;
 performTransition (state, t, i);
```

Other auxiliary procedures:

- Procedure initialStates (objl) creates the initial Markov state, by grouping the initial states of objects together, separated by commas.
- Procedure satisfyGuard (state, guard) checks if the guard is satisfied or not.
- Procedure replace (state, i, dst) returns state[i] = dst
- Procedure transitionExists(s, d, tl) checks if the new transition with source = s, destination = d, exists in the transition list or not.
- Procedure addNewState(n) add the new state n to the new state list nl, if it is neither in ol nor nl.
- Procedure isActionBefore (action) checks if the parameter in the action is "before" or "after".
- Procedure indexOfObjectByAction (action) first get the object name from the passed in parameter action, then returns the object position in the objects list.
- Procedure findTranByAction (i, action) first locate the trigger specified in the action, and then returns the transition that having this trigger.
- Procedure mark_goods (ol) marks the "up" states in the old list.

APPENDIX B

THE UNIFORMIZATION PROCEDURE

Algorithm: Uniformization Procedure

Given:

Q — infinitesimal generator matrix
π_0 — initial probability vector
ε — error tolerance
t — desired solution time
Δt — time interval length.

Variables:

P — discretized transition matrix
I — identity matrix
Γ — scaling factor
π — probability vector
i, j — iteration counter
k — series truncation point
l — time intervals

Procedure:

Chose $\Gamma \geq \max_i |q_{ii}|$

$$P = I + \frac{1}{\Gamma} Q$$

$$l = t/\Delta t$$

Determine the largest k with $1 - \sum_{i=0}^{k} e^{-\Gamma t} \frac{(\Gamma t)^i}{i!} \leq \varepsilon / l$

$\pi = \pi_0$
for $i = 1$ to l do {
    $\pi = \pi e^{-\Gamma t}$
    for $j = 1$ to k do {
        $\pi = \pi + \pi P \frac{\Gamma t}{j}$
    }
}

Vector-matrix-multiplication(π, P) Procedure

Assume:

I. P is implemented in column-wise vectors of size n, with only nonzero elements filled in, together with their position stored separately.

II. πP_i is the summation of all the nonzero elements of P_i multiplied by the corresponding elements in π.

for $i = 1$ to n do {
    $\pi_i = \pi P_i$
}

REFERENCES

[1] J.C. Laprie, "Dependability: a unifying concept for reliable computing and fault tolerance," *Dependability of Resilient Computers*, T. Anderson Ed., Blackwell Scientific Publications Professional Books, 1989, pp.1-28.

[2] G. Booch, J. Rumbaugh, and I. Jacobson, *The unified modeling language user guide*, Addison Wesley, 1999.

[3] K. Goseva-Popstojanova, A. Hassan, A. Guedem, W. Abdelmoez, D. Nassar, H. Ammar, and A. Mili, "Architectural-level risk analysis using UML," *IEEE Transactions on Software Engineering*, vol. 29, no. 10, Oct., 2003, pp. 946-960.

[4] T. Wang, A. Hassan, A. Guedem, W. Abdelmoez, K. Goseva-Popstojanova, and H. Ammar, "Architectural level risk assessment tool based on UML specifications," *Proceedings of the Twenty-fifth International Conference on Software Engineering* (ICSE03), Portland, Oregon, 2003.

[5] I. Majzik, A. Pataricza, and A. Bondavalli, "Stochastic dependability analysis of system architecture based on UML models", In R. De Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems, LNCS 2677*, Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, New York, 2003, pp. 219-244.

[6] G. Huszerl and I. Majzik, "Modeling and analysis of redundancy management in distributed object-oriented system by using UML statecharts," *Proceedings of the Twenty-seventh Euromicro Conference*, Warsaw, Poland, Sept. 4-6, 2001. pp. 200-207.

[7] G. J. Pai and J. B. Dugan, "Automatic synthesis of dynamic fault trees from UML system models," *Proceedings of the Thirteenth International Symposium on Software Reliability Engineering (ISSRE'02),* Annapolis, Maryland, Nov., 2002.

[8] M. Boger, M. Jeckle, S. Müller, and J. Fransson, "Diagram interchange for UML," *Proceedings of the 5th International Conference on The Unified Modeling Language,* Dresden, Germany, Sept. 30-Oct. 4, 2002.

[9] W. McUmber and B. Cheng, "A general framework for formalizing UML with formal languages," *Proceedings of the Twenty-third International Conference on Software Engineering (ICSE01),* Toronto, Ontario, Canada, May 12 – 19, 2001.

[10] D. Milićev, "Automatic model transformations using extended UML object diagrams in modeling environments," *IEEE Transactions. on Software Engineering*, vol. 28, no. 4, April, 2002, pp. 413-431.

[11] A. Johnson and M. Malek, "Survey of software tools for evaluating reliability, availability, and serviceability," *ACM Computing Surveys*, vol. 20, no. 4, December, 1988, pp. 227-269.

[12] K. S. Trivedi and M. Malhotra, "Reliability and performability techniques and tools: a survey," *Proceedings of Seventh ITG/GI Conference, MMB, Aachen* University of Technology, Sept. 27 - 48, 1993.

[13] J. Huang and M. J. Zuo, "Generalized multi-state k-out-of-n:G systems," *IEEE Transaction on Reliability*, vol. 49, no. 1, March, 2000, pp. 105-111.

[14] S. V. Amari, H. Pham, and G Dill, "Optimal design of k-out-of-n:G subsystems subjected to imperfect fault-coverage," *IEEE Transaction on Reliability*, vol. 53, no. 4, December, 2004.

[15] J. A. Abraham, "An improved algorithm for network reliability," *IEEE Transactions on Reliability*, vol. 28, no. 1, April, 1979, pp. 58-61.

[16] M. Veeraraghavan and K. S. Trivedi, "An improved algorithm for symbolic reliability analysis," *IEEE Transaction on Reliability*, vol. 40, no. 3, Aug. 1991, pp 347-358.

[17] S. Rai, M. Veeraraghavan, and K. S. Trivedi, "A survey of efficient reliability computation using disjoint products approach," *Networks*, vol. 25, May, 1995, pp. 147-163.

[18] T. Luo and K. S. Trivedi, "An improved algorithm for coherent-system reliability," *IEEE Transaction on Reliability*, vol. 47, no.1, March, 1998, pp.73-78.

[19] A. O. Balan, and L. Traldi, "Preprocessing minpaths for sum of disjoint products," *IEEE Transactions on Reliability*, vol. 52, no. 3, Sept, 2003, pp. 289-295.

[20] J. B. Dugan and K. S. Trivedi, "Coverage modeling for dependability analysis of fault tolerant systems," *IEEE Transactions on Computers*, vol. 38, no. 6, June, 1989, pp. 775-787.

[21] F. P. Mathur, "Automation of reliability evaluation procedures through CARE-the computer-aided reliability estimation program," *Proceedings of the AFIPS Fall Joint Computer Conference*, vol. 41, 1972, pp. 65-82.

[22] J. B. Dugan, K. S. Trivedi, M. K. Smotherman, and R. M. Geist, "The hybrid automated reliability predictor," *AIAA J. Guidance, Control, and Dynamics*, vol. 9, no. 3, May-June, 1986, pp. 319-331.

[23] C. Hirel, R. A. Sahner, X. Zang, and K. S. Trivedi, "Reliability and performability modeling using SHARPE 2000," *Proceedings of the Eleventh International Conference on Computer Performance Evaluation: Modeling Techniques and Tools*, March 27 - 31, 2000.

[24] C. Hirel, B. Tuffin, and K. S. Trivedi, "SPNP: stochastic petri nets. Version 6.0," *Proceedings of the Eleventh International Conference on Computer Performance Evaluation: Modeling Techniques and Tools*, 2000.

[25] G. Chiola, G. Franceschinis and R. Gaeta, M. Ribaudo, "GreatSPN 1.7: graphical editor and analyzer for timed and stochastic Petri Nets," *Performance Evaluation*, 24:47–68, 1995.

[26] A. Cumani, "ESP - A package for the evaluation of stochastic Petri nets with phase-type distributed transition times," In *Proceedings International Workshop Timed Petri Nets*, Torino, Italy, 1985, pp. 144-151.

[27] D. D. Deavours, W. D. Obal, M.A. Qureshi, W.H. Sanders, and A. van Moorsel, "UltraSAN version 3 overview," *Sixth International Workshop on Petri Nets and Performance Models (PNPM '95),* Durham, North Carolina, USA, Oct. 03 - 06, 1995.

[28] K.J. Sullivan, J.B. Dugan, and D. Coppit, "The Galileo fault tree analysis tool," In *Proceedings of the Twenty-ninth Annual International Symposium on Fault-Tolerant Computing (FTCS'99)*, IEEE Computer Society Press, Pittsburg, 1999, pp. 232-235.

[29] J. T. Blake and K. S. Trivedi, "Reliability analysis of interconnection networks using hierarchical composition," *IEEE Transactions on Reliability*, vol. 38, no.1, pp. 111-119, April, 1989.

[30] R. A. Sahner and K. S. Trivedi, "A hierarchical, combinatorial-Markov model of solving complex reliability models," Proceedings of 1986 ACM Fall joint computer conference, Dallas, Texas, United States, pp. 817–825.

[31] Kishor S. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications.* John Wiley & Sons, Inc., New York, 2002.

[32] J. Muppala, M. Malhotra, and K. S. Trivedi, "Markov dependability models of complex systems: analysis techniques," *Reliability and Maintenance of Complex Systems*, S. Ozekici (ed.), pp. 442-486, Springer-Verlag, Berlin, 1996.

[33] O. C. Ibe, R. C Howe, and K. S. Trivedi, "Approximate availability analysis of VAXcluster systems," *IEEE Transactions on Reliability*, vol. 38, no.1, April, 1989, pp. 146-152.

[34] 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.

[35] R. M. Smith, *Markov Reward Models: Application Domains and Solution Methods*, PhD dissertation, Department of Computer Science, Duke University, 1987.

[36] S. Berson, E. Souza e Silva, and R. Muntz, "A methodology for specification and generation of Markov models," in *Proceedings of the First Int. Conf. Numerical Solution of Markov Chains,* Raleigh, NC, Jan., 1990.

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

[38] A. Puliafito, M. Telek, and K. S. Trivedi, "The evolution of stochastic Petri Nets," Proceedings of the World Congress on Systems Simulation (WCSS '97), Singapore, Sept. 1-3, 1997.

[39] J. B. Dugan, *Extended Stochastic Petri Nets: Applications and Analysis*, PhD dissertation, Department of Computer Science, Duke University, 1984.

[40] M. Malhotra and K. S. Trivedi, "Dependability modeling using Petri-Nets," *IEEE Transaction on Reliability,* vol. 44, no. 3, Sept., 1995, pp. 428-440.

[41] G. Ciardo, A. Blakemore, P. F. J. 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 (eds. C. Meyer and R. J. Plemmons), Heidelberg: Springer-Verlag, 1993, pp. 141-191.

[42] J. B. Dugan, S. J. Bavuso, and M. A. Boyd, "Dynamic fault-trees models for fault tolerant computer systems," *IEEE Transactions on Reliability,* vol. 41, no. 3, Sept., 1992, pp. 363-373.

[43] M. A. Boyd, *Dynamic fault-trees models: techniques for analysis of advanced fault tolerant computer systems*, PhD dissertation, Department of Computer Science, Duke University, 1991.

[44] T. Sterling, *Beowulf cluster computing with Linux,* Cambridge, Massachusetts MIT Press, 2002.

[45] D. J. Becker, T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawak, and C. V. Packer, "Beowulf: a parallel workstation for scientific computation," *Proceedings of International Conference on Parallel Processing*, 1995.

[46] T. Naughton, S. L. Scott, Y. Fang, P. Pfeiffer, B. Ligneris, and C. Leangsuksun, "The OSCAR toolkit: current and future developments," *Dell Power Solutions*, Nov., 2001.

[47] C. Leangsuksun, L. Shen, T. Liu, H. Song, and S. L. Scott, "Availability prediction and modeling of high availability OSCAR cluster," *IEEE International Conference on Cluster Computing*, Hong Kong, December 1-4, 2003.

[48] M. Lanus, L. Yin, and K. S. Trivedi, "Hierarchical composition and aggregation of state-based availability and performability models," *IEEE Transactions on Reliability*, vol. 52, no.1, March, 2003, pp. 44-52.

[49] J. Lee, S. J. Chapin and S. Taylor, "Reliable heterogeneous applications," *IEEE Transactions on Reliability*, vol. 52, no. 3, Sept. 2003, pp. 330-339.

[50] P. Bremaud, *Markov Chains: Gibbs Fields, Monte Carlo Simulation, and Queues.* Springer, 1999.

[51] M. Iosifescu, *Finite Markov Processes and Their Applications*, John Wiley and Sons, 1980.

[52] A. Riska, *Aggregate matrix-analytic techniques and their applications*, PhD dissertation, Department of Computer Science, College of William & Mary, 2003.

[53] G. Strang, *Introduction to applied mathematics*, Wellesley-Cambridge Press, Cambridge, 1986.

[54] W. Stewart, *Introduction to the numerical solution of Markov Chains,* Princeton University Press, Princeton, 1994.

[55] R. Sidje and W. Stewart, "A numerical study of large sparse matrix exponentials arising in Markov Chains," *Computational Statistics & Data Analysis*, vol. 29, no. 3, Jan. 1999, pp. 345-368.

[56] B. Philippe, Y. Saad, and W. Stewart, "*Numerical methods in Markov Chain modeling,"* Operations Research, vol. 40, no. 6, 1992, pp. 1156-1179.

[57] A. Reibman and K.S. Trivedi, "Numerical transient analysis of Markov models," *Computer Operations Research,* vol. 15, no. 1, 1988, pp. 19-36.

[58] M. Malhotra, J. Muppula, and K. S. Trivedi, "Stiffness-tolerant methods for transient analysis of stiff Markov Chains," *Microelectronics and Reliability*, vol. 34, no. 11, 1994, pp. 1825-1841.

[59] C. Lindemann, M. Malhotra, and K. S. Trivedi, "Numerical methods for reliability evaluation of Markovian closed Fault-tolerant systems," *IEEE Transactions on Reliability*, vol. 44, no. 4, 1995, pp. 694-704.

[60] R.J. Boucherie and E.A. van Doorn, "Uniformization for lambda-positive Markov chains," *Stochastic Models*, vol. 14, 1998, pp. 171-186.

[61] B. L. Fox, P. W. Glynn, "Computing Poisson probabilities," *Communications of the ACM*, vol. 31, no. 4, 1988, pp. 440-445.

[62] E. de Souza e Silva and R. Gail, "Calculating cumulative operational time distributions of repairable computer systems," *IEEE Transactions on Computer*, vol. 35, 1986, pp. 322-332.

[63] D. Gross and D. R. Miller, "The randomization technique as a modeling tool and solution procedure for transient Markov processes," *Operation Research*, vol. 32, no. 2, 1984, pp. 343-361.

[64] D. Jin and S. G. Ziavras, "A super-programming technique for large sparse matrix multiplication on PC clusters," *IEICE Transactions on Information and Systems, Special Issue on Hardware/Software Support for High Performance Scientific and Engineering Computing*, vol. E87-D, no. 7, July 2004.

[65] Linpack, http://www.netlib.org/linpack/.

[66] JMP - A sparse matrix library for Java, http://www.math.uib.no/~bjornoh/jmp/index2.html

[67] B. Venners, *Inside The Java Virtual Machine*, McGraw-Hill, 1997.

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

[69] D. M. Daly, *Bounded aggregation techniques to solve large Markov models*, Ph.D. dissertation, Department of Electrical Engineering, University of Illinois at Urbana-Champaign, 2005.

[70] J. L. Devore, *Probability and statistics*, the Fifth ed., Brook/Core, New York, 2000.

[71] M. H. DeGroot and M. J. Schervish, *Probability and Statistics*, the Third ed., Addition-Wesley, New York, 2002.

[72] R. V. Hogg and A. T. Craig, *Introduction to Mathematical Statistics, the Fourth ed.*, Macmillan Publishing Co., Inc., New York, 1978.

[73] T. Y. Lin and D. P. Siewiorek, "Error log analysis: statistical modeling and heuristic trend analysis," *IEEE Transactions on Reliability*, vol. 39, no. 4, Oct. 1990, pp. 419-432.

[74] A. Wein and A. Sathaye, "Validating complex computer system availability models," *IEEE Transactions on Reliability*, vol. 39, no. 4, Oct. 1990, pp.468-479.

[75] D. Tong and K. Iyer, "Dependability measurement and modeling of a multicomputer system," *IEEE Transactions on Computers*, vol. 42, no. 1, Jan, 1993, pp. 62-75.

[76] R. Chillarege, S. Biyani, and J. Rosenthal, "Measurement of failure rate in widely distributed software," *Proceedings of the Twenty-fifth International Symposium on Fault-Tolerant Computing (FTCS 25)*, Sendai, Japan, 1996.

[77] P. Moran, P. Gaffney, J. Melody, M. Condon and M. Hayden, "System availability monitoring," *IEEE Transactions on Reliability*, vol. 39, no. 4, Oct., 1990, pp. 480-485

[78] T. Mattson, B. Sanders, and B. Massingill, *Patterns for Parallel Programming*, Addison-Wesley, 2005.

[79] H. Song, C. Leangsuksun, N. Gottumukkala, R. Nassar, S. L. Scott, and Andy Yoo, "Near-Real-time Availability Monitoring and Modeling for HPC/HEC runtime systems," *Symposium of Los Alamos Computer Science Institute,* Santa Fe, New Mexico, October, 2005.

[80] Y. Liu, C. Leangsuksun, H. Song, and S. L. Scott, "Reliability-aware Checkpoint /Restart Scheme: A Performability Trade-off," *IEEE International Conference on Cluster Computing*, 2005.

[81] H. Song, C. Leangsuksun, and R. Nassar, "OOMSE – An Object Oriented Markov Chain Specification and Evaluation Framework," *The Seventeenth International Conference on Software Engineering and Knowledge Engineering*, Taipei, Taiwan, 2005.

[82] H. Song, C. Leangsuksun, R. Nassar, Y. Liu, C. Engelmann, and S. L. Scott, "UML-based Beowulf Cluster Availability Modeling," *International Conference on Software Engineering Research and Practice*, Las Vegas, Nevada, 2005.

[83] H. Song, C. Leangsuksun, R. Nassar, and Y. Liu "Availability Specification and Evaluation of HA-OSCAR Servers – An Object-Oriented Approach," *The Third International Conference on Computing, Communications and Control Technologies*, Austin, Texas, July, 2005.

[84] H. Song, C. Leangsuksun, and R. Nassar, "A Light-Weight Solution for Large Sparse Markov Processes," *Proceedings of the Forty-third ACM Southeast Conference,* Kennesaw, Georgia, March 18-20, 2005.

[85] H. Song and C. Leangsuksun, "A Framework for Cluster Availability Specification and Evaluation," *Proceedings of the Forty-third ACM Southeast Conference,* Kennesaw, Georgia, March 18-20, 2005.

[86] C. Leangsuksun, C. Kottapalli, H. Song, and Y. Liu, "Reliability-Aware Intelligent Checkpointing of MPI Programs in Beowulf Clusters," *High Availability and Performance Computing Workshop,* Santa Fe, New Mexico, 2004.

[87] C. Leangsuksun, L. Shen, T. Liu, H. Song and S. L. Scott, "Availability Prediction and Modeling of High Availability OSCAR Cluster," *IEEE International Conference on Cluster Computing*, Hong Kong, December, 2003.

[88] C. Leangsuksun, L. Shen, T. Liu, H. Song and S. L. Scott, "Dependability Prediction of High Availability OSCAR Cluster Server," *Proceeding of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, June, 2003.

[89] C. Leangsuksun, H. Song, and L. Shen, "Reliability Modeling Using UML," *Proceeding of the International Conference on Software Engineering Research and Practice*, Las Vegas, Nevada, June, 2003.

[90] C. Leangsuksun, L. Shen, H. Song, S. L. Scott, and I. Haddad, "The Modeling and Dependability Analysis of High Availability OSCAR Cluster System," *Proceeding of The Seventeenth Annual International Symposium on High Performance Computing Systems and Applications*, Sherbrooke, Canada, May, 2003.