Availability Modeling and Evaluation on High Performance Cluster Computing Systems

Hertong Song, Chokchai Leangsuksun and Raja Nassar

College of Engineering & Science
Louisiana Tech University
Ruston, LA 71270, USA
hsong99@yahoo.com, {box, nassar}@latech.edu

Cluster computing has been attracting more and more attention from both the industrial and the academic world for its enormous computing power 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 paper, 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. The paper also presents a monitoring and data analysis framework, which is responsible for failure analysis and availability reconfiguration.

ACM Classification: D.2.11, D.2.12, D.2.13

1. INTRODUCTION

Cluster computing is becoming popular for its enormous computational power. High availability features need to be included 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-offs and refinements. The early evaluation of system characteristics, such as dependability (Laprie, 1989), 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 (Blake and Trivedi, 1989; Trivedi, 2002), Markov chains (Muppala *et al*, 1996; Ibe *et al*, 1989; Haverkort and Trivedi, 1993), Stochastic Petri Net (SPN) (Puliafito *et al*, 1997; Dugan, 1984), 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-

Manuscript received: 19 December 2005
Communicating Editor: Hassan Reza

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, and SPN has places and transitions. As a consequence, Markov chain and Petri Net models are often large when the systems are complicated. These large models may be beyond the intuition of modelers, lose the logic view of the system, and 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 (Haverkort and Trivedi, 1993; Johnson and Malek, 1998; Trivedi and Malhotra, 1993; Berson *et al*, 1990). 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 (Booch *et al*, 1999). UML, which is used to model software systems, can be extended to model hardware systems as well, since it provides features such as stereotypes, tagged values, and constraints that can be customized and extended. With embedded tag-value pairs, UML can be utilized to model systems' availability aspects.

Upon the availability model is created, there are needs to find solutions of the model. Therefore, the modeling evaluation is incorporated into the framework, and it is responsible to solve the availability model correctly and give the result. Moreover, the system's health needs to be monitored once it is on the running state. This is to ensure the system's runtime availability meets its design goal, and helps to identify the cause of the trouble. Our approach includes three major aspects, namely (1) the modeling, (2) model evaluation, and (3) monitoring and analysis. Figure 1 shows the 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 able to be updated during the runtime that facilitates the dynamic monitoring and analysis. An object-oriented modeling scheme (Song *et al*, 2005a) 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, are given by formulas, while others such as Markov model need numerical solutions (Stewart, 1994; Song *et al*, 2005b; Lindemann *et al*, 1995; Boucherie and van Doorn, 1998; Fox and Glynn, 1988; de Souza e Silva and Gail, 1986; Gross and Miller, 1984).

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.

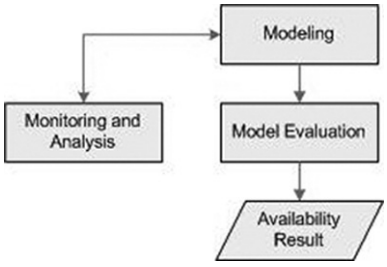


Figure 1: The Modeling and evaluation framework

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. Then it passes this information to the evaluation facility to reevaluate the system's availability.

The rest of the paper is organized as follows: Section 2 lays out the availability modeling concepts. Section 3 presents the model decomposition and a variety of availability models. The object-oriented Markov model specification is illustrated in Section 4. The monitoring and analysis facility is discussed in Section 5 followed by the conclusion and future work

2. BACKGROUND

In this section, we introduce some background information concerning the evaluation of availability to fault tolerance systems. We first give a brief overview of the analytical models, which is categorized into combinatorial model, Markov models and other models deviated from Markov models. Then we describe the terminologies of fault, error and failure.

2.1 Combinatorial Models

Analytical models are mathematical models which are an abstraction from the real world system and relate only to the behaviour and characteristics of interest. Examples of analytical models are combinatorial models, Markov models, Petri Nets, and hierarchical models (Sahner *et al*, 1996). Each type of these models requires different solution techniques.

Combinatorial models capture the static behaviour of the system being modeled. The solutions of these types of models are simply series-parallel reliability computations. Reliability block diagram (RBD), fault tree (FT), reliability graph are in this category. They are similar in the way that they capture conditions that makes a system fail in terms of structural relationships between the system components. The solution of a combinatorial model involves three steps: (1) a set of minimal paths (minpath) is generated, (2) make sure that all the paths are pair wise disjoint, and (3) apply the series-parallel formula. Algorithms for generating disjoint minpath are Abraham (1979), Veeraraghavan and Trivedi (1991), Rai *et al* (1995), Luo and Trivedi (1998), Balan and Traldi (2003). The major difference of these algorithms is that of using single variable inversion or multiple variable variables inversion.

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 Trivedi (2002).

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

and its instantaneous unavailability $U(t)$ is

$$U(t) = 1 - A(t) = \frac{\lambda}{\mu + \lambda} - \frac{\lambda}{\mu + \lambda} e^{-(\mu+\lambda)t} \tag{2}$$

2.2 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

behaviour because of their combinatorial characteristics. To address this type of system behaviour, Markov models have been popular techniques applied to a variety of fields. A Markov process (Trivedi, 2002; Bremaud, 1999; Iosifescu, 1980), which is a special case of stochastic processes whose dynamic behaviour is such that the probability distributions for its future development depend only on the present 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. Only a handful of Markov models can have closed form solutions, and the majority need numerical solutions (Stewart, 1994).

Markov reward model (Haverkort and Trivedi, 1993) is a Markov model with the reward assigned to each state and transitions, and each submodel is linked by mathematical expressions.

Petri Nets (Puliafito *et al*, 1997) 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 behaviours 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), stochastic reward nets (SRN), etc. (Puliafito *et al*, 1997; Dugan, 1984; Johnson and Malek, 1988).

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, HARP, and SHARP (Johnson and Malek, 1988; Trivedi and Malhotra, 1993).

Dynamic Fault Tree (DFT) uses the fault tree representation to capture systems dynamic behaviours by adding several logical gates (Dugan *et al*, 1992), 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.

2.3 Fault, Error, and Failure

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

An error is an undesired system behaviour that the system is not able to deliver services complying with what is expected of the system. An error is a manifestation of a fault.

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 2 shows the relationship of the three.

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

Figure 2: Fault, failure and error

3. MODELING AND SPECIFICATION

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 detailed interactions, then the resulting availability model could be very large. Thus, we adopt the "hierarchical composition" technique (Blake and Trivedi, 1989; Sahner and Trivedi, 1986; Lanus *et al*, 2003; Lee *et al*, 2003). The system availability model is divided into two submodels based on the functionalities of subsystems, a server submodel and a client submodel. The availability mode can be described by using a RBD (Reliability Block Diagram), as shown in Figure 3. Figure 3 shows a cluster system that requires a quorum of computers to be functioning. Certainly, this reliability block diagram can be extended by adding in more submodels if more facilities are considered likely to fail, such as network connections etc.

The system fails when either the server submodel or the client submodel fails. Thus, the system's availability is given by

$$A_{sys} = A_S A_N \tag{3}$$

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

$$A_N = \sum_{i=k}^{n} \binom{n}{i} A^i \overline{A}^{n-i} \tag{4}$$

where A and $\overline{A}$ are the availability and unavailability of a single client node at time t, given by Sahner *et al* (1996).

$$\overline{A}(t) = \frac{\lambda}{\lambda+\mu} - \frac{\lambda}{\lambda+\mu} e^{-(\lambda+\mu)t} \tag{5}$$

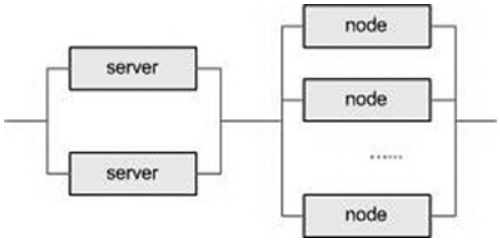


Figure 3: The RBD of cluster system availability model

$$A(t) = 1 - \overline{A}(t) \tag{6}$$

On the other hand, servers are normally implemented with complex mechanisms, led to a complicated model which needs to describe more intricate interactions. The complicated behaviour can be modeled by a continuous time Markov chain (Blake and Trivedi, 1989; Trivedi, 1989; Muppala *et al*, 1996; Sahner and Trivedi, 1986). We adopt an object-oriented, event generating, and message passing technique (Berson *et al*, 1990) 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 which are described in the following section.

4. MARKOV MODEL SPECIFICATION AND GENERATION

4.1 Overview

An object has a unique event based behaviour 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.

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 5 gives a simple example of two objects' interaction.

Figure 5 shows a computing system with primary server P and a standby server S in two statechart diagrams. Figure 5 (a) shows primary server and Figure 5 (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 *t1* 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

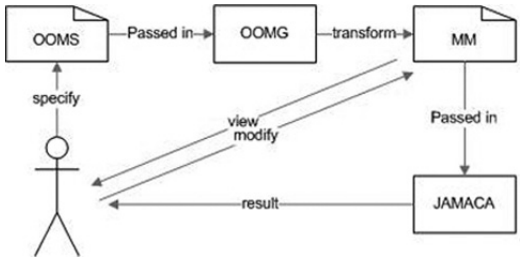


Figure 4: OOMSE Framework

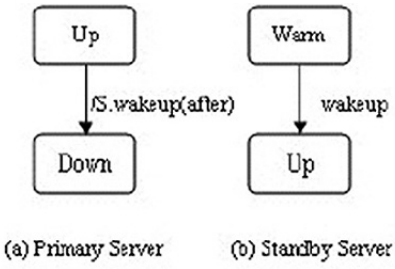


Figure 5: Statecharts for two servers

in the time of `t2`. The word "after" given in the parenthesis denotes that the action is taken after the transition. The sequence of Markov states is UW $\xrightarrow{t1}$ DW $\xrightarrow{t2}$ 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.

4.2 Definitions

Definition 1 An object is a 4-tuple $0 = \{N,S,S_0,E\}$ 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
- 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, $T = \{t_1,t_2,\ldots t_{|T|}\}$.

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 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, $A = \{a_1,a_2,\ldots a_{|A|}\}$

4.3 Grammar

The grammar (Aho *et al*, 1986) 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→ *STS*|*ST*
ST→ *String*
TGN→ *String*
Op→ [&&| ||]
String→ [a-zA-Z0-9]$^{+}$
Num→ [0-9]$^{+}$.[0-9]$^{+}$
Terminals = <, >, /, systemup, objects, object, name, states, state, initial events, transition, src, dst, rate, trigger, guard, action, before, after.

4.4 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].

4.4.1 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;
```

4.4.2 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.

4.4.3 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".

4.4.4 Perform Transition

The perform transition procedure first creates a new state by replacing the current state's ith 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.

```
<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>
```

Table 1: OOMS of two servers in XML

4.4.5 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.

4.5 Example

We adopt the HA-OSCAR system (Leangsuksun *et al*, 2003) 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 "woken 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 1 gives the XML specification for the two servers.

In Table 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 is 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 2 and Table 3 list the generated Markov states and transitions.

5. MONITORING AND ANALYSIS

In this section, we discuss an availability-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. There is a wide variety of research that is based upon the analysis of event logs. Lin and Siewiorek (1990) analyze the error log file on file servers to demonstrate the log is composed of at least transient and

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

Table 2:
Markov states 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

Table 3: Markov transitions for two servers

intermittent processes. Wein and Sathaye (1990) present their experience with validation of complex computer system availability models. Tong and Iyer (1993) measures the failure rate in widely distributed software. Chillarege *et al* (1996) presented a failure rate measurement technique on distributed software, based upon classifying failure data into "failure windows." Moran *et al* (1990) 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. 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 resources. 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.

5.1 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 shows the reliability-aware monitoring and modeling 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

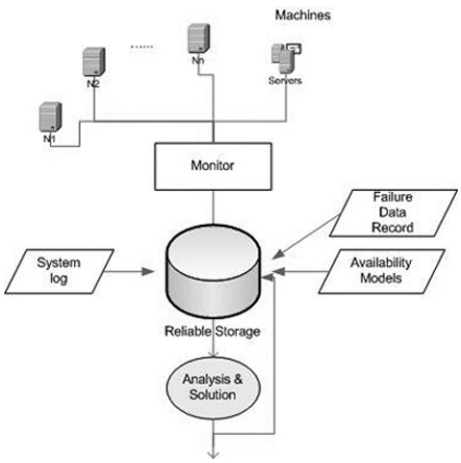


Figure 6: Monitoring and analysis framework

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 reliable disk storage. The Analysis and 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 7 shows the flow diagram inside the monitoring and analysis framework.

In Figure 7, 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. 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

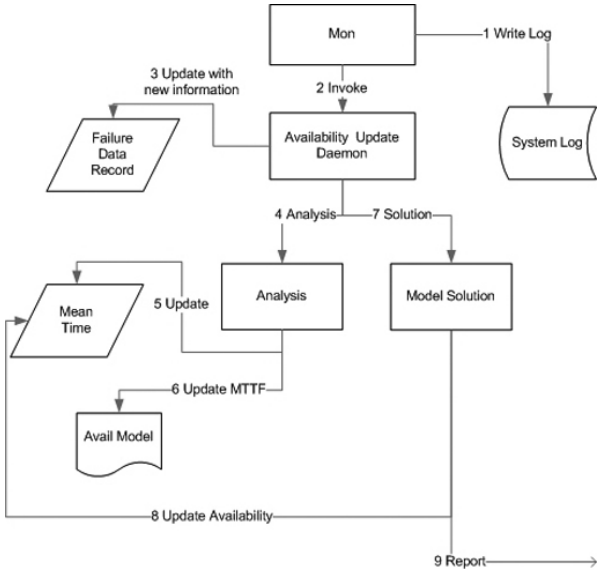


Figure 7: Monitoring and analysis flow diagram

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 this information to the solution engine to have the availability result. 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.

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.

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.

5.2 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 past years, 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.

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 8 Availability of each node in the White system the availability for each node in the White cluster system.

From Figure 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 9 and Figure 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 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 10, node downtime density indicates that the most of the total downtime for each node are around 100 hours; some failure

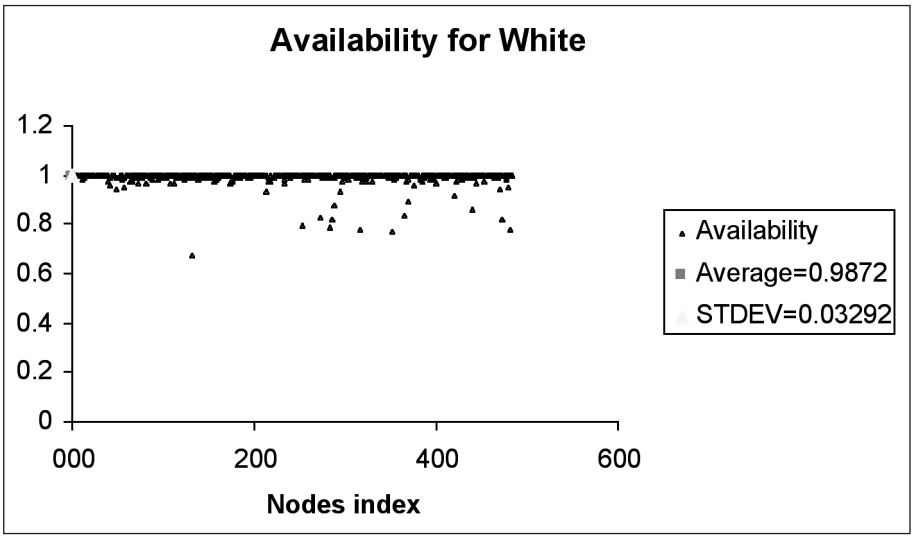


Figure 8: Availability of each node in the White system

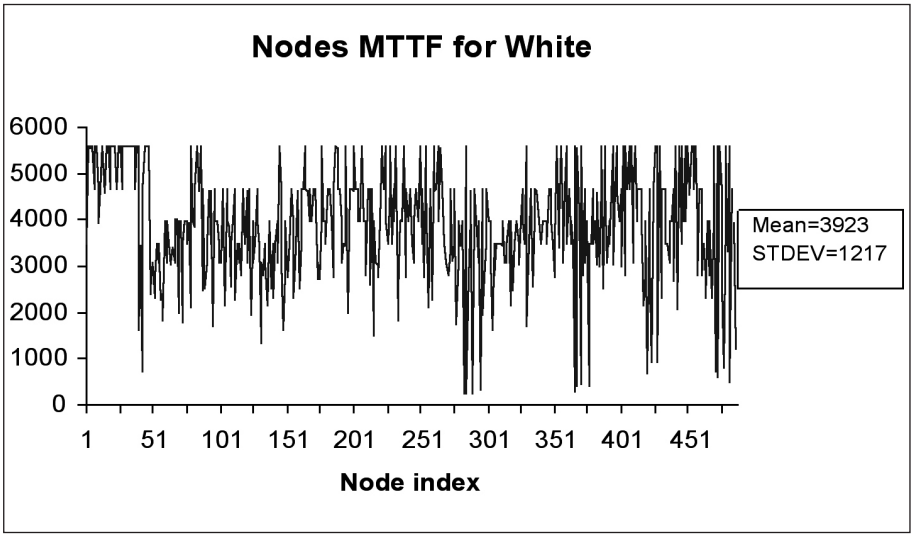


Figure 9: Nodes MTTF density

events cost more time to be fixed, thus increasing the total average TDT. Figure 11 shows the average downtime per failure for each node in the cluster White. The mean is 27.92 hours, and the standard deviation is 85. From these figures, we can see that some failures took much longer time to be recovered than the majority.

6. CONCLUSION AND FUTURE WORK

This paper presents a novel technique to facilitate the availability modeling, runtime monitoring, and near-real-time availability evaluation. The background and basic concepts of modeling

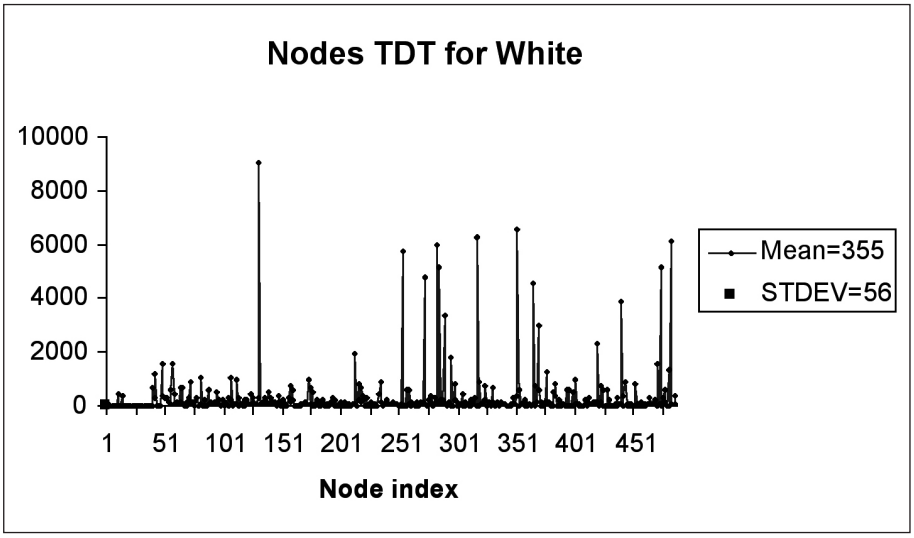


Figure 10: Node downtime (in hours)

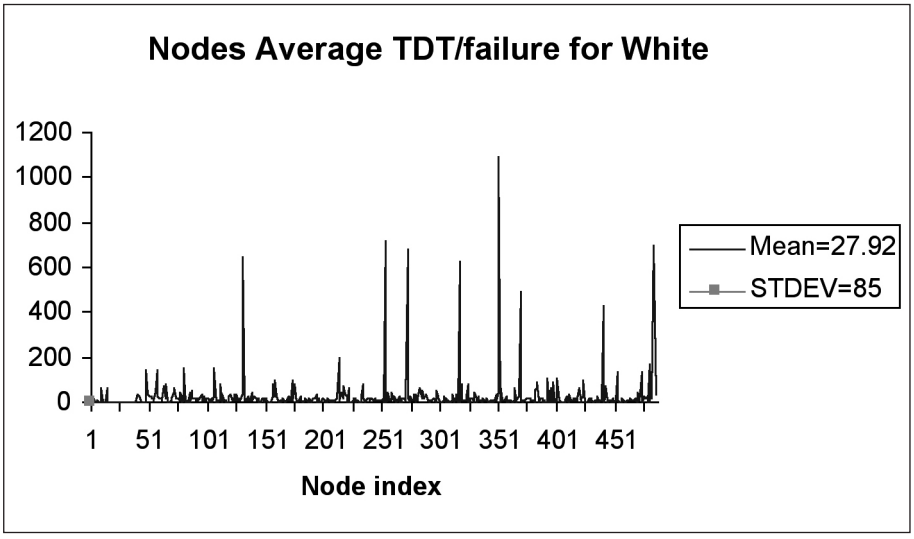


Figure 11: The average downtime per failure

techniques were reviewed first. We then characterized HPC system modeling as our targeted problem domain. Three key components are described in the paper, 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.

In the monitoring and analysis section, we presented a framework to enable automatic data analysis and availability update. This framework is not only 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.

Currently, we consider each computer as a single instance under the assumption each instance has 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 (Bremaud, 1999), 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 behaviours, such as software failure. The monitoring and analysis facility needs to be able to diagnose 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 that, if the monitoring cannot receive the response of any node, it considers that the node too 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 (Trivedi, 2002; Bremaud, 1999; Iosifescu, 1980) should also be investigated to represent the system's behaviours.

REFERENCES:

ABRAHAM, J. A. (1979): An improved algorithm for network reliability, *IEEE Trans. on Reliability*, 28(1): 58-61, April.

AHO, A. V., SETHI, R. and ULLMAN, J. D. (1986): *Compilers: Principles, Techniques, and Tools*, Addison-Wesley.

BALAN, A. O. and TRALDI, L. (2003): Preprocessing minpaths for sum of disjoint products, *IEEE Trans. on Reliability*, 52(3): 289–295, September.

BERSON, S., DE SOUZA E SILVA, E. and MUNTZ, R. (1990): A methodology for specification and generation of Markov models, in *Proc. 1st Int. Conf. Numerical Solution of Markov Chains*. (Raleigh, NC), January.

BLAKE, J. T. and TRIVEDI, K. S. (1989): Reliability analysis of interconnection networks using hierarchical composition, *IEEE Trans. on Reliability*, 38(1): 111–119, April.

BOGER, M., JECKLE, M., MÜLLER, S. and FRANSSON, J. (2002): Diagram interchange for UML, *Proceedings of the 5th International Conference on The Unified Modeling Language*, Dresden, Germany, September 30–October 4.

BOOCH, G., RUMBAUGH, J. and JACOBSON, I. (1999): *The Unified Modeling Language User Guide*, Addison Wesley.

BOUCHERIE, R.J. and VAN DOORN, E.A. (1998): Uniformization for lambda-positive Markov chains, Stochastic Models, 14: 171–186.

BREMAUD, P. (1999): *Markov Chains: Gibbs Fields, Monte Carlo Simulation, and Queues*. Springer.

CHILLAREGE, R., BIYANI, S. and ROSENTHAL, J. (1996): Measurement of failure rate in widely distributed software, *Proceedings of the Twenty-fifth International Symposium on Fault-Tolerant Computing (FTCS 25)*, Sendai, Japan.

DEGROOT, M. H. and SCHERVISH, M. J. (2002): *Probability and Statistics*, the Third ed., Addition-Wesley, New York.

DE SOUZA E SILVA, E. and GAIL, R. (1986): Calculating cumulative operational time distributions of repairable computer systems, *IEEE Transactions on Computer*, 35: 322–332.

DUGAN, J. B. (1984): *Extended Stochastic Petri Nets: Applications and Analysis*, PhD dissertation, Department of Computer Science, Duke University.

DUGAN, J. B., BAVUSO, S. J. and BOYD, M. A. (1992): Dynamic fault-trees models for fault tolerant computer systems, *IEEE Transactions on Reliability*, 41(3): 363–373, September.

FOX, B. L. and GLYNN, P. W. (1988): Computing poisson probabilities, *Communications of the ACM*, 31(4): 440–445.

GOSEVA-POPSTOJANOVA, K., HASSAN, A., GUEDEM, A., ABDELMOEZ, W., NASSAR, D., AMMAR, H. and MILI, A. (2003): Architectural-level risk analysis using UML, *IEEE Transactions on Software Engineering*, 29(10): 946–960, October.

GROSS, D. and MILLER, D. R. (1984): The randomization technique as a modeling tool and solution procedure for transient Markov processes, *Operation Research*, 32(2): 343–361.

HAVERKORT, B. R. and TRIVEDI, K. S. (1993): Specification and generation of Markov reward models, Discrete-event dynamic systems: Theory and Applications, 3: 219–247.

HUSZERL, G. and MAJZIK, I. (2001): Modeling and analysis of redundancy management in distributed object-oriented system by using UML statecharts, *Proceedings of the Twenty-seventh Euromicro Conference*, Warsaw, Poland, September 4–6, 200-207.

IBE, O. C., HOWE, R.C. and TRIVEDI, K.S. (1989): Approximate availability analysis of VAXcluster systems, *IEEE Trans. on Reliability*, 38(1): 146–152, April.

IOSIFESCU, M. (1980): *Finite Markov Processes and Their Applications*, John Wiley and Sons.

JOHNSON, A. and MALEK, M. (1988): Survey of software tools for evaluating reliability, availability, and serviceability, *ACM Computing Surveys*, 20(4): 227–269, December.

LANUS, M., YIN, L. and TRIVEDI, K. S. (2003): Hierarchical composition and aggregation of state-based availability and performability models, *IEEE Trans. on Reliability*, 52(1): 44–52 March.

LAPRIE, J.C. (1989): Dependability: A unifying concept for reliable computing and fault tolerance, *Dependability of Resilient Computers*, ANDERSON, T. Ed., Blackwell Scientific Publications Professional Books, 1–28.

LEANGSUKSUN, C., SHEN, L., LIU, T., SONG, H. and SCOTT, S. L. (2003): Availability prediction and modeling of high availability OSCAR cluster, *IEEE International Conference on Cluster Computing*, December 1–4, Hong Kong.

LEE, J., CHAPIN, S. J. and TAYLOR, S. (2003): Reliable heterogeneous applications, *IEEE Trans. on Reliability*, 52(3): 330–339, September.

LIN, T. Y. and SIEWIOREK, D. P. (1990): Error log analysis: statistical modeling and heuristic trend analysis, *IEEE Transactions on Reliability*, 39(4): 419–432, October.

LINDEMANN, C., MALHOTRA, M. and TRIVEDI, K. S. (1995): Numerical methods for reliability evaluation of Markovian closed fault-tolerant systems, *IEEE Transactions on Reliability*, 44(4): 694–704.

LIU, Y., LEANGSUKSUN, C., SONG, H. and SCOT, S. L. (2005): Reliability-aware checkpoint /restart scheme: A performability trade-off, *IEEE International Conference on Cluster Computing*.

LUO, T. and TRIVEDI, K. S. (1998): An improved algorithm for coherent-system reliability, *IEEE Trans. on Reliability*, 47(1): 73–78, March.

MAJZIK, I., PATARICZA, A. and BONDAVALLI, A. (2003): Stochastic dependability analysis of system architecture based on UML models, In De LEMOS, R., GACEK, C. and ROMANOVSKY, A. editors, *Architecting Dependable Systems, LNCS 2677*, Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, New York, 219–244.

McUMBER, W. and CHENG, B. (2001): 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.

MILICEV, D. (2002): Automatic model transformations using extended UML object diagrams in modeling environments, *IEEE Transactions. on Software Engineering*, 28(4): 413–431, April.

MORAN, P., GAFFNEY, P., MELODY, J., CONDON, M. and HAYDEN, M. (1990): System availability monitoring, *IEEE Transactions on Reliability*, 39(4): 480–485, October.

MUPPALA, J., MALHOTRA, M. and TRIVEDI, K. S. (1996): Markov dependability models of complex systems: Analysis techniques, *Reliability and Maintenance of Complex Systems*, OZEKICI, S. (ed.), 442–486, Springer-Verlag, Berlin.

PAI, G. J. and DUGAN, J. B. (2002): Automatic synthesis of dynamic fault trees from UML system models, *Proceedings of the Thirteenth International Symposium on Software Reliability Engineering (ISSRE'02)*, Annapolis, Maryland, November.

PULIAFITO, A., TELEK, M. and TRIVEDI, K. S. (1997): The evolution of Stochastic Petri Nets, *Proc. World Congress on Systems Simulation (WCSS '97)*, Singapore, September 1–3.

RAI, S., VEERARAGHAVAN, M. and TRIVEDI, K. S. (1995): A survey of efficient reliability computation using disjoint products approach, *Networks*, 25: 147–163, May.

SAHNER, R. A. and TRIVEDI, K. S. (1986): A hierarchical, combinatorial-Markov model of solving complex reliability models, *Proceedings of 1986 ACM Fall joint computer conference*, Dallas, Texas, United States, 817–825.

SAHNER, R.A., TRIVEDI, K.S. and PULIAFITO, A. (1996): *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package*, Kluwer Academic Publishers, New York.

SONG, H., LEANGSUKSUN, C., GOTTUMUKKALA, N., NASSAR, R., SCOTT, S. L. and YOO, A. (2005): Near-real-time availability monitoring and modeling for HPC/HEC runtime systems, *Symposium of Los Alamos Computer Science Institute*, Santa Fe, New Mexico, October.

SONG, H., LEANGSUKSUN, C. and NASSAR, R. (2005a): OOMSE – An object oriented Markov chain specification and evaluation framework, *The 17th International Conference on Software Engineering and Knowledge Engineering*, Taipei, Taiwan.

SONG, H., LEANGSUKSUN, C. and NASSAR, R. (2005b): A light-weight solution for large sparse Markov processes, *Proceedings of the Forty-third ACM Southeast Conference*, Kennesaw, Georgia, March 18–20.

STEWART, W. (1994): *Introduction to the numerical solution of Markov Chains*, Princeton University Press, Princeton.

TONG, D. and IYER, K. (1993): Dependability measurement and modeling of a multicomputer system, *IEEE Transactions on Computers*, 42(1): 62–75, January.

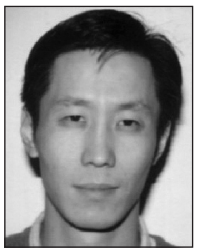
TRIVEDI, K.S. (2002): *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. John Wiley & Sons, Inc. New York.

TRIVEDI, K. S. and MALHOTRA, M. (1993): Reliability and performability techniques and tools: A survey, *Proceedings of 7th ITG/GI Conference*, MMB, Aachen University of Technology, September, 27–48.

VEERARAGHAVAN, M. and TRIVEDI, K. S. (1991): An improved algorithm for symbolic reliability analysis, *IEEE Trans. on Reliability*, 40(3): 347–358, August.

WEIN, A. and SATHAYE, A. (1990): Validating complex computer system availability models, *IEEE Transactions on Reliability*, 39(4): 468–479, October.

BIOGRAPHICAL NOTES



Hertong Song

Hertong Song holds a PhD in Computational Analysis and Modeling, and a Master of Mathematics from Louisiana Tech University. He has a Master of Computer Science from the University of Akron and a Bachelor of Engineering from Tsinghua University. He has worked as a software engineer in both Telecomm and in the financial area. His research interests include Distributed Computing, Availability Analysis, and Service-Oriented Architecture.

Chokchai Leangsuksun

Dr Chokchai "Box" Leangsuksun is an associate professor in computer science at the Center for Entrepreneurship and Information Technology (CEnIT) at Louisiana Tech University. He received the PhD and MSc in computer science from Kent State University, Kent, Ohio in 1989 and 1995 respectively. His research interests include Highly Reliable and High Performance Computing, Intelligent Component Based Software Engineering, Parallel & Distributed Computing, Service-Oriented Architecture, Service engineering and management.

Raja Nassar

Dr Raja Nassar is a Maxfield Endowed Professor of Mathematics and Statistics at Louisiana Tech University. He has contributed to hundreds of publications in the field of Applied Statistics. His research interests include Applied Statistics, Applied Probability and Stochastic Processes, Statistical and Mathematical Modeling, Applied Mathematics, Design and Data Analysis.