CONSTRUCTION AND SOLUTION OF MARKOV REWARD MODELS

by

Muhammad Akber Qureshi

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 9 6

Just to increase a page number.

STATEMENT BY AUTHOR

To my parents and my wife Afshan

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Stochastic Petri nets (SPNs) and extensions are a popular method for evaluating a wide variety of systems. In most cases, the interesting measures regarding the system's characteristics can be defined at the net level by means of reward variables. Depending on the measures, these net-level reward models are solved either by first generating a state-level reward model or by directly generating paths from the net-level description. In this thesis, we propose algorithms for the generation of state-level reward models as well as for directly obtaining solutions from net-level reward models when the net-level reward models are specified as stochastic activity networks (SANs) with "step-based reward structure".

Moreover, we propose algorithms for computing the expected value and the probability distribution function of a reward variable at specified time instants, and for computing the probability distribution function of reward accumulated during a finite interval. The interval may correspond to the mission period in a mission-critical system, the time between scheduled maintenances, or a warranty period; whereas the time instants may be critical instances during these intervals. The proposed algorithms avoid the construction of state-level representations and the memory growth problems experienced when applying previous approaches to large models.

Furthermore, we study the effect of workload on the availability and response time of voting algorithms. Voting algorithms are a popular way to provide data consistency

in replicated data systems. Many models have been made to study the degree to which replication increases the availability of data, and some have been made to study the cost incurred in maintaining consistency. However, little work has been done to evaluate the time it takes to serve a request, accounting for server and network failures, or to determine the effect of workload on these measures. In this thesis, we use stochastic activity networks (SANs) to study the effect of work load on availability and mean response time of two variant models of a replicated file system to maintain data consistency, one using a static voting algorithm, the other using a dynamic voting algorithm.

CHAPTER 1

Introduction

1.1 Background

In the last decade, model-based performance and dependability evaluation has emerged as an integral part of the design process of computer and communication systems. Given the growing human dependence on these systems, they must have high performance and reliability. Modeling is one of the methods for evaluating competing design decisions. Creating a model is easier than building a prototype; it usually costs less than the prototype, and it reduces the evaluation effort by focusing attention on the system characteristics that are significant with respect to the performance measures of interest.

Modeling can be divided into three basic steps. The first step consists of building a stochastic representation of the system. The stochastic representation can be either a stochastic process or a specification formalism through which the stochastic process can be generated. The stochastic representation captures the transition behavior of the system. The second step defines performance/dependability measures by associating numerical gains or losses, directly or indirectly (through specification language), to the states and the transitions of the stochastic process. The third step solves for measures by either first generating the stochastic process from the specification and then solving it or by directly acting on the specification.

My research contributions are in developing a more general framework to associate gains or losses to components of stochastic activity networks (SANs) [1, 2], a specification formalism based on Petri nets, and in developing efficient and numerical stable techniques to solve for the transient behavior of Markov processes. The solution techniques developed in this dissertation are generally applicable, but the framework to associate gains or losses is specific to the SANs.

1.2 Overview

Due to the versatility and complexity of today's computer systems, the use of stochastic Petri nets (SPNs) and their extensions is becoming wide spread in the modeling community. Compared to queuing networks and state-level representations, SPNs have the advantage that many attributes of modern computer systems and networks (including contention for multiple resources, fault-tolerance, degradable performance, and timing constraints) can be represented precisely, and in a single model [3, 4].

We use stochastic activity networks (SANs), an extension to SPNs, as our modeling paradigm. In addition to the primary constructs of the SPNs, SANs provide "gates" and "cases." *Gates* enable the modeler to include algorithms and protocols related to the behavior of the system as C-language code. *Cases* enable the modeler to specify probabilistic choices the system can make after completion of events. Moreover, SANs use a model construction technique called *reduced base model construction* [5], which significantly reduces the state space generated when modeling hierarchical systems.

A SAN model represents the behavior of the system but does not specify what a modeler wants to know about the system. One may be interested in studying characteristics related to performance such as throughput, response time, availability, reliability, etc. SANs use "reward functions," first defined by Howard [6] at the stochastic process level, to specify such useful measures of performance. *Reward functions* are classified as *rate reward functions*, which assign time-dependent values to state occupancies, and *impulse reward functions*, which assign time-independent values to state transitions. Collectively, a stochastic process with its reward functions is classified as a *state-level reward model.*

In the same way, reward functions are defined at the SAN level. Specifically, SANs define an "activity-marking" oriented reward structure. In an *activity-marking* reward structure, rate reward functions are associated with the possible markings of the SANs which constitute the state space of the underlying stochastic process, and impulse reward functions are associated with activities which take the SAN from one marking to another marking. Other extensions of SPNs, for example generalized stochastic Petri nets (GSPNs) [7] and stochastic reward nets (SRNs) [8], also define activity-marking reward structures. In addition to assigning impulse reward functions to completion of activities, Ciardo *et al.* [8] associate them with events such as completion of activities in particular markings. However, the definition of rate reward functions is identical in SANs, GSPNs, and SRNs.

Traditionally, a solution for the specified reward variable is obtained by first constructing a stochastic process representation from the net-level model (e.g., [2, 8, 9]) and then solving the constructed process for the desired variable. A stochastic process is constructed by exploring the reachability graph and storing the set of reachable states. Often, the amount of memory required to store the reachability set (due to the large number of states) makes it impossible to construct the stochastic process. To overcome this problem of generation of large state spaces, two approaches have been proposed. The first approach (as discussed in [10, 11, 12, 13]) is based on the observation that most of the time computer systems and networks reside in a small subset of states and, resultingly, these states contain most of the probability mass. Therefore, for certain interesting measures, a solution can be obtained within a reasonable error bound by generating just these highly probable states. Unfortunately, in most cases, it is not possible to generate the high-probable states without generating a substantial portion of the state space.

The second approach (as discussed in [14, 15, 16]) avoids the storage problem related to the generation of large state spaces by directly analyzing the paths (sequences of states) in the reachability graph. In the path-based approach, only one path is required to be stored in the memory at a time, but this savings comes at the cost of additional computation (i.e., compared to generating a reachability set, since a state is explored as many times as the number of different paths can reach it). This trade of storage complexity for time complexity is advantageous if probability mass is concentrated on a reasonably small number of paths (tens of millions), which is often the case for Markov processes representing computer systems and networks. Another factor which favors this approach is that computers are becoming faster at a higher rate than the rate at which memory capacity is increasing.

Depending on the distributions assigned to the activities at the net level, different types of stochastic behaviors can result from the SPN specification. Ciardo *et al.* [17] distinguish these underlying stochastic behaviors as Markov, semi-Markov, semi-regenerative, and generalized semi-Markov. These stochastic behaviors combined with the reward functions form reward models. A reward model quantifies the behavior of the system with respect to the performance variable but does not specify the type of solution, and hence the variable in which the modeler is interested. For example, if the modeler is interested in availability, then both steady-state and transient solutions can be obtained. Moreover, within transient analysis two types of solutions are possible, time-averaged interval-of-time (time-averaged value of the accumulated reward over the mission time) and instant-of-time (value of the reward variable at instances during the mission time) [18]. If the reward model quantifies reliability, then only an instant-of-time solution makes sense. (A classification of different types of reward variables depending on the types of solutions can be found in [18].)

In this thesis, one focus is on developing efficient techniques for transient solution of Markov reward models. Particularly, we are interested in instant-of-time and interval-of-time solutions.

1.2.1 Instant-of-Time Measures

A measure which depends on the state of a process at a specific instant during its evolution is classified as an instant-of-time measure. An instant-of-time measure is evaluated by computing the state-occupancy probabilities of states at the specific instant. Reliability is a common dependability variable which can be represented in this way. It requires the computation of the state-occupancy probability of the system's failed state (which is made to be an absorbing state) at the desired instant-of-time. Suppose f is the failed state and $\pi_f(t)$ is its occupancy probability at time t, then reliability $R(t)$ of the system can be computed as $(1 - \pi_f(t))$. (Note the way reliability is computed in the modeling community, it can be classified as an instant-of-time measure. However, it is an interval-of-time variable and is defined as the probability of a system to provide uninterrupted service over an interval.)

Generally, computation of instant-of-time variables requires computing the state-occupancy probability vector $\pi(t) = [\pi_1(t), \pi_2(t), \ldots, \pi_M(t)]$, where M is the number of states in the process and $\pi_i(t)$ is the state-occupancy probability of state $i \in M$. For a continuous-time homogeneous Markov process with generator matrix Q, according to Kolmogorov forward equations [19], the state-occupancy probability vector $\pi(t)$ obeys

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

whose general solution is given by

$$\pi(t) = \pi(0)e^{Qt}. \tag{1.1}$$

Many numerical methods have been proposed to solve Equation 1.1 (for a survey see [20, 21, 22, 23]). Most of these methods use techniques such as matrix powering, matrix decomposition, ordinary differential equations (ODEs), or uniformization. Matrix powering and decomposition techniques very often suffer from the ill-conditioning of the generator matrix. ODE techniques have the problem that there is no one ODE technique that can outperform all others for all values of t, generator matrix, and required accuracy. Uniformization [24, 25, 26] (also known as Jenson's method and randomization) is very stable, easy to program, and outperforms other methods in most cases, but its performance degrades considerably with an increase in the stiffness of the system. (Stiffness is related to large differences in magnitude of various state transition rates and to large values of instant-of-time t relative to the largest state departure rate.)

Methods based on matrix powering, ODEs, and uniformization show individual shortcomings, but collectively provide a numerically stable and computationally efficient way of computing the state-occupancy probability vector $\pi(t)$. Furthermore, their solutions are limited to solving systems up to only a few hundred thousands states, since require the knowledge of the complete generator matrix. Resultingly, they require that the state-space be constructed prior to their application. As discussed earlier, state-space construction is a memory-complex problem. Given a reasonable amount of memory (64-128 Mbytes), we can construct systems with few hundred-thousand states. Obviously, with more memory, one can generate larger processes, but there are limits.

In the context of the solution of instant-of-time variables, two approaches have been proposed to avoid the memory complexity. The first approach [10, 11, 12, 13] suggests that rather than constructing the complete state space, only states with high probability mass be generated. This approach has the problem that even if the set of high-probable states is small, in most of the cases, it is not possible to generate this set without generating a substantial number of states of the state space. The second approach suggests [15, 16] that, instead of states, the analysis should be done on the paths (state-trajectories of the underlying Markov process). The path-based approach avoids the memory problem but at an additional computational cost, since there can be infinitely many more paths in a stochastic process than the number of states. The computational cost of the approach is reasonable if the probability mass concentrates on a reasonable number of paths (tens of millions). Therefore, potentially, a path-based approach can provide solutions for large state-space systems with a reasonable number of high-probability paths.

An important issue in the path-based approach is how to compute the probability of a path. Two different approaches have been used in this regard. In [15], Bon and Collet have proposed an approach based on a renewal argument. Their approach is limited since it only allows paths which consist of states whose sums of outgoing rates are distinct.

In [16], Nicol and Palumbo have suggested uniformization as an approach to compute path probabilities. Uniformization is numerically stable, and allows consideration of paths without distinct outgoing rates, but can be computationally very expensive because the Poisson process, which controls the transitions of the subordinated Markov chain, must have a rate corresponding to the maximum departure rate of all states in the path.

As argued in the preceding discussion, despite of all the advances made in solving instant-of-time variables, we are still limited in solving small-sized models consisting of hundreds of thousands of states. In some methods, the limitation is due to the memory complexity, while in other methods, the limition is due to their computational complexity.

In the next section, we present an overview of the methods that have been developed to solve the probability distribution of interval-of-time variables.

1.2.2 Interval-of-Time Measures

A measure which depends on the accumulated behavior of a process over the utilization interval is classified as an interval-of-time variable. Interval availability is the most commonly measured interval-of-time variable in dependability evaluation. It gives the total amount of time over the utilization interval for which the system remains operational. A more meaningful measure, *time-averaged interval of time*, can be obtained by dividing the interval-of-time variable by the length of the interval. A time-averaged interval availability measure gives the fraction of the utilization interval for which the system remains operational.

Performability variables can often be expressed as interval-of-time variables. Performability evaluation [27] strives to predict the "bottom-line" performance of a computing system accounting for possible degradation of performance due to faults. Many methods have been proposed to evaluate system performability, but one of the most popular has been through the use of reward models [28]. In the reward model context, performability is defined as the probability distribution function of reward accumulated during a finite interval.

Most early work in solving such models has been limited to acyclic Markov reward models (see, for example, [28, 29, 30, 31, 32]). Determining the distribution of reward accumulated over a finite interval for general (possibly cyclic) Markov reward models is more difficult due to the possible presence of an infinite number of paths. The first attempt to solve such models was made by Kulkarni *et al.* [33]. They numerically inverted an expression obtained in the transform domain to obtain a solution. Later Smith *et al.* [34] presented an improved version of the algorithm in [33], with a computational complexity of $O(n^3)$, where n is the number of states. In [35], Pattipati *et al.* applied partial differential equation techniques to compute system performability.

Also notable is the work of de Souza e Silva and Gail, who were the first to present a performability model solution based on uniformization [36]. They formulated the probability distribution of reward accumulated over a finite interval by first conditioning on the number of transitions n of a uniformized process and then further conditioning on vectors corresponding to the number of visits to states with different rewards, given n. This algorithm was limited in applicability since the storage and computational complexity was combinatorial with the number of distinct rewards. Two attempts have been made to deal with this complexity. In particular, Donatiello and Grassi [37] presented an algorithm with a polynomial complexity in number of distinct rewards by combining uniformization and Laplace transform methods. In addition, in [38], de Souza e Silva and Gail presented a significant improvement on their previous algorithm, albeit limited to rate-based reward models. The storage complexity of their new algorithm is independent of the number of distinct rewards assigned to the states of the Markov process.

In spite of these significant advances, the developed algorithms have been limited to solving small models, and for short time intervals. The reasons are two-fold: 1) the storage complexity is still polynomial with number of states and transitions of the uniformized process, and 2) in addition to the stated storage complexity, the algorithm requires the storage of the state transition matrix of the entire subordinated Markov process. When state spaces are large (as is often the case) or the interval is reasonably long (again, as is typical), these storage requirements make the computation of the distribution of accumulated reward over a finite interval very difficult.

1.3 Research Objectives

Despite the past work just outlined, we are limited in specifying measures of interest that depend on individual events in the system, such as being in a particular state of the system or transitions between states of the system, and in solving small-sized Markov reward models that are not capable of modeling details of today's complex computer and communication systems. In view of these shortcomings, the rest of the section summarizes my research contributions in the area of reward variable specification and Markov reward model solution.

In previous attempts to define reward functions at the SPN level, the rate reward functions are defined on the markings of the SPNs, and the impulse reward functions are either defined on the completion of individual activities or completion of activities in particular markings. These reward functions are limited in defining performance measures since they are not capable of defining impulse reward functions on all individual events and sequences of events which can take a SPN from one marking (state) of the model (system) to another marking (state) of the model (system). Due to these constraints, at the reward specification level, we present:

- a very general "step-based" reward structure, which allows a modeler to assign impulse reward functions to all types of events that can take a SAN from one stable marking to another stable marking, and
- an algorithm to generate a state-level representation from a SAN specification and the associated step-based reward function.

At the specification step, the step-based reward structure extends the set of reward variables that can be specified at the net level. Reward variables can then be solved by either first constructing a state-level representation and then solving it or by directly solving the underlying reward model by generating the paths in the trajectory space. Both approaches are useful. The approach based on first constructing the state space is necessary for steady-state solutions and provides efficient and numerically stable transient solutions for small-sized systems (consisting of hundreds of thousands of states). Whereas the path-based approach can potentially provide transient solutions (instant-of-time and interval-of-time) of realistic sized models since they avoid the construction of a complete state space. As discussed earlier, algorithms based on path exploration have been proposed for instant-of-time measures, but they are either numerically unstable, computationally expensive (using uniformization to compute path probabilities), or limited in solving the types of systems (paths with all distinct rate rewards). In view of these shortcomings, for instant-of-time variables, we develop:

- a path-based numerically stable solution technique which directly works on the continuous-time Markov process by using convolution integration.

Computation of the distribution of interval-of-time variables (performability) is a more complex problem than the instant-of-time variables. In addition to more computation, it requires significantly more memory at the execution time. In particular, it requires memory to store: 1) the state transition matrix of the entire subordinated Markov process, and 2) vectors corresponding to the number of visits to states with different rewards. As discussed earlier, despite recent algorithmic advances, the storage complexity for reward vectors is still polynomial with the number of states and transitions of the uniformized process. When state spaces are large (as is often the case) or the interval is reasonably long (again, as is typical), both these storage requirements make the computation of distribution of accumulated reward over a finite interval very difficult. In view of these shortcomings, for interval-of-time variables, we develop:

- a path-based approach in which we also trade time for memory to compute the distribution of time-averaged reward accumulated over a finite interval for general (possibly cyclic) reward models, and

- a numerically stable and computationally efficient algorithm to compute the conditional distribution of accumulated reward given a path.

The new approach is significant in that it can be applied to much larger systems than previously possible, and it is numerically stable.

Moreover, I have worked on the development of the performance/dependability modeling tool *UltraSAN* [39]. An overview of *UltraSAN* is presented as an appendix. To show its usefulness, I have successfully used it to model and study

- the response time and availability of voting algorithms.

The remainder of the thesis is organized as follows. Chapter 2 presents the step-based reward structure, an algorithm to check the probabilistic completeness of a SAN-based reward model, and an algorithm to generate the state-level representation while eliminating the unstable markings on-the-fly.

Chapter 3 presents a numerically stable path-based solution for instant-of-time measures which works equally well for both acyclic and cyclic Markov processes. We use a renewal argument to compute the required path probabilities. Moreover, our solution is not restricted to paths which consist of states with distinct minimum holding times.

Chapter 4 presents a new methodology for calculating the distribution of reward accumulated over a finite interval. In particular, we derive recursive expressions for the distribution of reward accumulated given that a particular sequence of state changes occurs during the interval, and we explore paths one at a time. The utility of the methodology is illustrated via application to a realistic fault-tolerant multiprocessor model with over a half million states.

Chapter 5 presents the application of the *UltraSAN* modeling tool in the study of voting algorithms. In particular, it presents the study of the effect of varying workload on availability and response time of voting algorithms.

Chapter 6 summarizes the results.

At the end, as an appendix, a description of the performance/dependability modeling tool *UltraSAN* is included. *UltraSAN* is an X-window based software package which is used at more than 90 universities and several companies.

CHAPTER 2

Step-Based Reward Structure

Stochastic Petri nets (SPNs) and extensions can compactly represent the behavior of a wide variety of systems. They have the advantage that many attributes of modern computer systems and networks (including contention for multiple resources, fault-tolerance, degradable performance, and timing constraints) can be represented precisely and in a single model. Likewise, through the use of reward structures and variables defined at the network level [8, 18], one can define many useful measures of performance, dependability, and performability on the network representation. Most often, a solution of the specified reward variables is obtained by constructing a finite-state stochastic process representation from the net-level model (e.g., [2, 8, 9]) and then solving the constructed process for the desired variables.

In this chapter, we define a "step-based reward structure" which compared to the activity-marking reward structure allows us to specify a richer set of reward variables. Specifically, in addition to the reward variables that can be specified by an activity-marking reward structure, it allows us to specify reward variables by defining impulse rewards on sequences of events that can happen, taking a SAN from one marking to another. Moreover, an algorithm to construct a state-level reward model from a SAN with its associated step-based reward structure is also presented. The algorithm to generate a state-level representation includes a check to determine if the SAN is "well specified" and eliminates "unstable markings" on-the-fly, as each stable state is generated, rather than generating the entire reachability set first and then eliminating the unstable markings. On-the-fly elimination can save a significant amount of memory in the generation process, avoiding state generation problems that will occur if there are a very large number of unstable markings.

The remainder of the chapter is organized as follows. Section 2.1 gives an overview of SANs which will be helpful in understanding the step-based reward structure and the state generation procedure. Section 2.2 formally introduces the step-based reward structure, while section 2.3 illustrates its usefulness. Section 2.4 presents an algorithm that checks if a SAN and its associated step-based reward structure is well specified (i.e., completely probabilistically specified). Section 2.5 uses the well-specified check algorithm to configure the example introduced in section 2.3. Finally, Section 2.6 presents an algorithm to construct a state-level reward model.

2.1 Review of SAN Terminology

2.1.1 SAN Primitives

Stochastic activity networks are a stochastic extension to activity networks [2]. Informally, SANs consist of *places*, *activities*, *input gates*, and *output gates*. Places hold tokens, as in Petri nets. Collectively, the distribution of tokens among all the places defines the *marking* of the SAN. Let P be some finite set of places in a SAN. Formally, then, a marking is a mapping $\mu_p : P \rightarrow \mathbb{N}$. A set of possible markings of the SAN is the set of functions

$M_P = \{\mu_p | \mu_p : P \rightarrow I\!N\}$. If S is a set of places ($S \subset P$, $S \neq \emptyset$), a marking of S is a mapping $\mu_s : S \rightarrow I\!N$. μ_s is called a *partial marking* of the SAN. A set of possible markings of S is the set of functions $M_S = \{\mu_s | \mu_s : S \rightarrow I\!N\}$.

In a SAN model, an event in the system being modeled is represented by either a *timed* or an *instantaneous* activity. Each timed activity is assigned an activity time distribution function which probabilistically specifies its activity time. Activity time distributions can be generally distributed random variables [2]. Instantaneous activities, on the other hand, represent events which are completed in a negligible amount of time. Furthermore, each activity, timed or instantaneous, has a strictly positive number of cases attached to it. Cases associated with an activity provide probabilistic choices about the next marking the SAN will enter after completion of the activity. These probabilistic choices are defined by assigning a discrete probability distribution to the cases of each activity.

Input gates have a finite set of inputs and one output, where the inputs are places and the output is an activity. Each input gate has a computable predicate and function, which are both defined on the associated input places. If the predicate is true and the attached activity completes, the marking of the SAN is changed according to the defined function. In this way, the input gates can put the SAN directly into any desired marking, eliminating the need to build a subnetwork of places and instantaneous activities to implement the function. Output gates are provided for a similar reason and have one input and a finite set of outputs; the input is a case of an activity, while all outputs are places. A computable function is associated with each output gate. If the case attached to the output gate is chosen upon completion of an activity, its function is executed to change the marking of the SAN.

2.1.2 SAN Execution

The execution of a SAN is a characterization of the possible completions of activities, selections of cases, and changes in markings. The marking of a SAN can only change by completion of an enabled activity. An activity is *enabled* in a marking if all the directly connected places contain at least one token and, in the case of an input gate connected to the activity, the predicate of the input gate is true. An activity *may complete* in a marking, if it is enabled, and, if it is timed, there are no instantaneous activities enabled. This definition thus gives instantaneous activities priority over timed activities.

The "yields" function describes the marking change that occurs when an activity completes and a case is chosen. Formally, the notation $\mu \stackrel{a,c}{\rightarrow} \mu'$ is used to indicate that the completion of an activity a and choice of the case c *yield* $\mu^{'}$. The set of all the reachable markings from a given marking is defined in terms of the reflexive, transitive closure of the yield relation, $\stackrel{*}{\rightarrow}$. Resultingly, the set of all the reachable markings of a SAN $R(SAN, \mu_0)$, where μ_0 is the initial marking, is given by $R(SAN, \mu_0) = \{\mu | \mu_0 \stackrel{*}{\rightarrow} \mu\}$. Reachable markings are classified into *stable* and *unstable* markings. *Stable reachable markings* (denoted by the set $SR(SAN, \mu_0) \subseteq R(SAN, \mu_0)$) are those in which no instantaneous activity is enabled. *Unstable reachable markings* (denoted by $UR(SAN, \mu_0) \subseteq R(SAN, \mu_0)$) are those in which at least one instantaneous activity is enabled.

Furthermore, each event in the evolution of a SAN is defined by a marking-activity-case triple $<\mu, a, c>$, called a *configuration*, where a is an activity with case c, which may complete in μ. A completion of a configuration occurs when the activity a in marking μ completes by choosing case c. The execution of a SAN can be described in terms of sequences of configurations, more formally called "stable steps." Each stable step in the execution of a SAN takes it from one stable marking to a next stable marking.

Definition 2.1.2.1: A *stable step* in the execution of a SAN with initial marking μ_0 is a sequence of configurations,

$$< \mu_1, a_1, c_1 >< \mu_2, a_2, c_2 > \ldots < \mu_n, -, - >,$$

where $n \geq 2$, and

1. $\mu_1, \mu_n \in SR(SAN, \mu_0)$,
2. for $2 \leq i \leq n-1$, $\mu_i \in UR(SAN, \mu_0)$, and
3. for $1 \leq i \leq n-1$, $\mu_i \stackrel{a_i, c_i}{\rightarrow} \mu_{i+1}$.

Performance and dependability measures can be specified in terms of reward variables defined on SANs. In the next section, we will introduce a reward structure that includes and generalizes previously proposed ways of specifying reward variables.

2.2 Step-Based Reward Structure

Reward variables are an established method of specifying performance and dependability measures [18, 8]. In this formalism, rate rewards are associated with stable markings, and the reward is accumulated with the specified rate during the time the SAN is in the marking. Likewise, impulse rewards are assigned to events in the SAN execution, and are accumulated every time the particular event occurs. In their most general form, these events can be, for example, completions of activities or sequences of activities, choices of cases, and entrances to markings. This gives us great flexibility in defining measures, and makes it possible to capture sophisticated measures of worth which depend on times spend on markings, and benefits and costs assigned to events or sequences of events.

As will be seen below, events in a SAN execution can be mapped on stable steps and/or "partial steps" in the SAN execution. A *partial step* is a sequence of configurations in which the number of configurations is less than the number of configurations in a stable step. To support the definition of rewards on stable and partial steps, we therefore introduce in this section a so-called *step-based reward structure*. In this structure, impulse rewards which are assigned to an event on the SAN level may correspond to *sets* of stable or partial steps in the SAN execution. For instance, a reward assigned to an instantaneous activity will be part of any stable step that contains that activity in its sequence of configurations. A formalism to represent such sets based on the notion of stable, unstable, and special extended configurations follows. In particular, these extended configurations can be used to represent a set of partial steps where every element is either a configuration or is of type $<\mu, -, ->$. A "restricted sequence of extended configurations" can be used to represent a set of stable or partial steps. Formally,

Definition 2.2.1: A *stable extended configuration* is a triple $(\boldsymbol{\mu}, \boldsymbol{a}, \boldsymbol{c})$, where

- $\boldsymbol{\mu}$ is either $\star$, μ_s, or μ_p, where $\star$ is the set of all stable reachable markings, $\mu_s \in M_S$ is a set of stable reachable markings which satisfy the partial marking μ_s, and $\mu_p \in M_P$ is a particular stable reachable marking of the SAN,

- $\boldsymbol{a}$ is either $\star$, s_a, or a, where $\star$ is the set of all timed activities, s_a is a particular set of timed activities, and a is a specific timed activity,
- $\boldsymbol{c}$ is either $\star$, s_c, or c, where $\star$ is the set of all cases, s_c is a particular set of cases, and c is a specific case.

Definition 2.2.2: An *unstable extended configuration* is a triple $(\boldsymbol{\mu}, \boldsymbol{a}, \boldsymbol{c})$, where

- $\boldsymbol{\mu}$ is either $\star$, μ_s, or μ_p, where $\star$ is the set of all unstable reachable markings, $\mu_s \in M_S$ is a set of unstable reachable markings which satisfy the partial marking μ_s, and $\mu_p \in M_P$ is a particular unstable reachable marking of the SAN,
- $\boldsymbol{a}$ is either $\star$, s_a, or a, where $\star$ is the set of all instantaneous activities, s_a is a particular set of instantaneous activities, and a is a specific instantaneous activity,
- $\boldsymbol{c}$ is either $\star$, s_c, or c, where $\star$ is the set of all cases, s_c is a particular set of cases, and c is a specific case.

Definition 2.2.3: A *special extended configuration* $(\mu_p, -, -)$ or $(\mu_s, -, -)$, specifies the event that the SAN enters the reachable marking μ_p or a marking in the set of reachable markings which satisfy the partial marking μ_s, respectively.

Now, sets of stable steps can be defined in terms of restricted sequences of extended configurations. In particular,

Definition 2.2.4: A *restricted sequence of extended configurations* is the sequence

$$(\boldsymbol{\mu_1}, \boldsymbol{a_1}, \boldsymbol{c_1})(\boldsymbol{\mu_2}, \boldsymbol{a_2}, \boldsymbol{c_2}) \ldots (\boldsymbol{\mu_n}, \boldsymbol{a_n}, \boldsymbol{c_n})(\boldsymbol{\mu_{n+1}}, -, -)^{0,1},$$

where $n \geq 0$, $(\boldsymbol{\mu_1}, \boldsymbol{a_1}, \boldsymbol{c_1})$ can be either a stable or an unstable extended configuration and $(\boldsymbol{\mu_n}, \boldsymbol{a_n}, \boldsymbol{c_n}), i = 2, \ldots, n$ are unstable extended configurations. The superscript on the special element denotes the number of times it appears in the sequence.

Using the above formalism, a reward structure can be given for rate and impulse rewards, such that rate rewards are associated with stable markings, and impulse rewards are associated with events in the SAN execution. To this end, let the set SEC denote the set of all possible restricted sequences of extended configurations, and let PM be the set of all possible stable markings and partial markings. Given these definitions, the step-based reward structure is defined as follows.

Definition 2.2.5: A *step-based reward structure* is a pair of functions:

- $R : \mathrm{PM} \rightarrow \mathbb{R}$ where for $r \in \mathrm{PM}$, $R(r)$ is the rate of reward accumulated in marking r, if r is a stable reachable marking. If r is a partial marking, then $R(r)$ is the rate of reward accumulated while in stable reachable markings which satisfy the partial marking r.

- $C : \mathrm{SEC} \rightarrow \mathbb{R}$ where for $\tau \in \mathrm{SEC}$, $C(\tau)$ is the impulse reward obtained upon completion of any step (partial or stable) in the restricted sequence of extended configurations τ.

Compared to previous reward structures (e.g., [18, 8]), the step-based reward structure provides a more flexible way of specifying impulse rewards. In particular, previously defined reward structures allow impulse rewards to be associated with events such as completion of activities [18] or completion of activities in particular markings [8]. In a step-based reward structure, these events correspond to extended configurations $(\star, a, \star)$ and $(\mu, a, \star)$, respectively. Thus, a step-based reward structure covers all previous ways in which impulse rewards are assigned at the stochastic Petri net level. Moreover, the step-based reward structure allows one to specify other types of impulse rewards on the SAN level, since it allows the assignment of impulse rewards to all possible sequences of events that can be mapped to partial or stable steps.

2.3 Example

To illustrate the use of SANs and step-based reward structures in representing systems and performance measures, consider a service station which consists of a single server and a finite queue. Two types of jobs, job_1 and job_2, arrive at the service station independently of each other. If the queue is full, an arriving job immediately leaves without obtaining any service. If, however, there is space in the queue, the arriving job enters and is placed at the end of the queue.

The server is capable of servicing jobs in two service modes. These service modes depend on the type of the job at the head of the queue. If the first job in the queue is of type job_1, then the server gives this job exclusive access to the server, with some (job-type specific) deterministic service time. If, however, the first job in the queue is of type job_2, then the server simultaneously serves all jobs of type job_2 that are in the queue. After completion of a service, each remaining job in the queue either moves forward in the queue (with probability p) or becomes impatient and leaves the queue (with probability $(1-p)$). The probability p depends on the number of service periods remaining until the job can acquire the server.

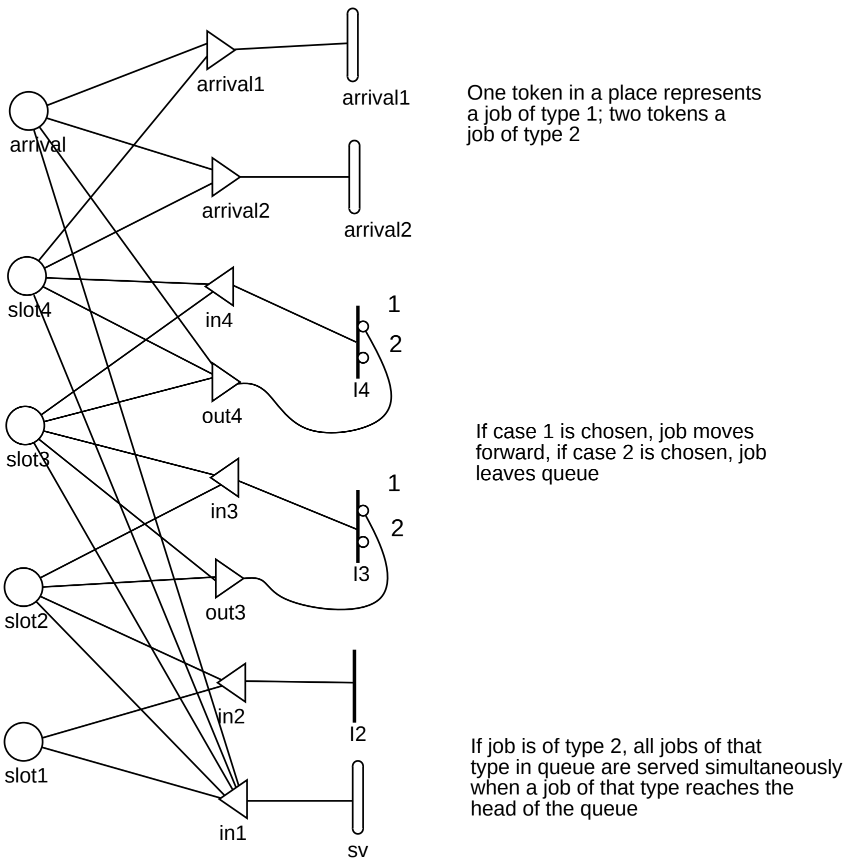

Figure 2.1: SAN example

Table 2.1: Input gate definitions for SAN example

Gate	*Definition*
in1	Predicate *MARK(slot1) > 0*
	Function *if (MARK(slot1) == 2) {* *if (MARK(slot4) == 2) MARK(slot4) = 0;* *if (MARK(slot3) == 2) MARK(slot3) = 0;* *if (MARK(slot2) == 2) MARK(slot2) = 0;* *}* *MARK(slot1) = 0;* *MARK(arrival) = 0;*
in2	Predicate *MARK(slot2) > 0 && MARK(slot1) == 0*
	Function *MARK(slot1) = MARK(slot2); MARK(slot2) = 0;*
in3	Predicate *MARK(slot3) > 0 && MARK(slot2) == 0*
	Function *identity*
in4	Predicate *MARK(slot4) > 0 && MARK(slot3) == 0*
	Function *identity*

Figure 2.1 and Tables 2.1, 2.2, 2.3, 2.4, and 2.5, collectively, depict the SAN model of the service station with capacity of four jobs. Places *slot1*, *slot2*, *slot3*, and *slot4* correspond to the first, second, third, and fourth position in the queue, respectively. One token in these places signifies that there is a job of type job_1 in the corresponding position, while two tokens signify a job of type job_2.

The timed activity *sv* and the input gate *in1* model the two service modes of the server. The server takes a different amount of deterministic time to perform each type of service. Hence, the timed activity *sv* has an activity time which depends on the number of tokens in the place *slot1*. After the server completes a service of type job_1, the job from the place *slot1* leaves the system (as specified by gate *in1*). However, when the server completes a

Table 2.2: Output gate definitions for SAN example

Gate	*Definition*
arrival1	*if (MARK(slot4) == 0) MARK(slot4) = 1;* *MARK(arrival) = 1;*
arrival2	*if (MARK(slot4) == 0) MARK(slot4) = 2;* *MARK(arrival) = 1;*
out3	*MARK(slot2) = MARK(slot3); MARK(slot3) = 0;*
out4	*MARK(slot3) = MARK(slot4); MARK(slot4) = 0;*

service of type job_2, the job from the place *slot1* leaves the system with other jobs of type job_2 (again as specified by gate *in1*). At this point, each remaining job can either move ahead in the queue with probability p or leave the system without getting service with probability $(1-p)$. Input gate *in4*, instantaneous activity *I4*, and output gate *out4* model this event for the job in place $slot4$. In this case, (marking-dependent) probability p is associated with *case1* of the activity *I4*, while probability $(1-p)$ is associated with *case2*, which represents leaving of the job from the queue. Similarly, input gate *in3*, instantaneous activity *I3*, and output gate *out3* model the shift or departure of the job in place $slot3$. The job in place $slot2$ always moves ahead in the queue since it is always the next job to be serviced. Note that after the completion of service, shifts and departures are completed in zero time, since *I2*, *I3*, and *I4* are instantaneous activities.

The two timed activities *arrival1* and *arrival2* model the arrival of jobs of type job_1 and job_2, respectively, at the service station. Both of these timed activities have exponentially distributed activity times. After the completion of either of these timed activities, the connected output gate, *arrival1* or *arrival2*, discards the arriving job if the queue is full. Otherwise, the job enters at the fourth position, *slot4*, in the queue. The input gate *in4*, instantaneous activity *I4*, and output gate *out4* shift the job from the fourth position to the

Table 2.3: Activity case probabilities for SAN example

<table>
<tr><th>Activity</th><th>Case</th><th>Probability</th></tr>
<tr><td rowspan="2">I3</td><td>1</td><td>if (MARK(arrival) == 0) {
if ((MARK(slot1) == 1) || (MARK(slot3) == 1 &&
MARK(slot1) == 2)) {
return(0.75);
}
else if ((MARK(slot1) == 0) || (MARK(slot3) == 2 &&
MARK(slot1) == 2)) {
return(1.0);
}
}
else {
return(1.0);
}</td></tr>
<tr><td>2</td><td>if (MARK(arrival) == 0) {
if ((MARK(slot1) == 1) || (MARK(slot3) == 1 &&
MARK(slot1) == 2)) {
return(0.25);
}
else if ((MARK(slot1) == 0) || (MARK(slot3) == 2 &&
MARK(slot1) == 2)) {
return(ZERO);
}
}
else {
return(ZERO);
}</td></tr>
<tr><td>I4</td><td>1</td><td>if (MARK(arrival) == 0) {
if ((MARK(slot1) == 1 && MARK(slot2) == 1) ||
(MARK(slot4) == 1 && MARK(slot1) == 1 &&
MARK(slot2) == 2) || (MARK(slot4) == 1 &&
MARK(slot1) == 2 && MARK(slot2) == 1)) {
return(0.5);
}
else if ((MARK(slot4) == 2 && MARK(slot1) == 1 &&
MARK(slot2) == 2) || (MARK(slot4) == 2 &&
MARK(slot1) == 2 && MARK(slot2) == 1) ||
(MARK(slot4) == 1 && MARK(slot1) > 0 &&
MARK(slot2) == 0) || (MARK(slot4) == 1 &&
MARK(slot1) == 0 && MARK(slot2) > 0) ||
(MARK(slot4) == 2 && MARK(slot1) == 1 &&
MARK(slot2) == 0) || (MARK(slot4) == 2 &&
MARK(slot1) == 0 && MARK(slot2) == 1) ||</td></tr>
</table>

Table 2.4: Activity case probabilities for SAN example

<table>
<tr><th>Activity</th><th>Case</th><th>Probability</th></tr>
<tr><td></td><td></td><td>(MARK(slot4) == 1 && MARK(slot1) == 2 &&
MARK(slot2) == 2)) {
return(0.75);
}
else if ((MARK(slot1) == 0 && MARK(slot2) == 0) ||
(MARK(slot4) == 2 && MARK(slot1) == 0 &&
MARK(slot2) == 2) || (MARK(slot4) == 2 &&
MARK(slot1) == 2 && MARK(slot2) == 0) ||
(MARK(slot4) == 2 && MARK(slot1) == 2 &&
MARK(slot2) == 2)) {
return(1.0);
}
}
else {
return(1.0);
}</td></tr>
<tr><td></td><td>2</td><td>if (MARK(arrival) == 0) {
if ((MARK(slot1) == 1 && MARK(slot2) == 1) ||
(MARK(slot4) == 1 && MARK(slot1) == 1 &&
MARK(slot2) == 2) || (MARK(slot4) == 1 &&
MARK(slot1) == 2 && MARK(slot2) == 1)) {
return(0.5);
}
else if ((MARK(slot4) == 2 && MARK(slot1) == 1 &&
MARK(slot2) == 2) || (MARK(slot4) == 2 &&
MARK(slot1) == 2 && MARK(slot2) == 1) ||
(MARK(slot4) == 1 && MARK(slot1) > 0 &&
MARK(slot2) == 0) || (MARK(slot4) == 1 &&
MARK(slot1) == 0 && MARK(slot2) > 0) ||
(MARK(slot4) == 2 && MARK(slot1) == 1 &&
MARK(slot2) == 0) || (MARK(slot4) == 2 &&
MARK(slot1) == 0 && MARK(slot2) == 1) ||
(MARK(slot4) == 1 && MARK(slot1) == 2 &&
MARK(slot2) == 2)) {
return(0.25);
}
else if ((MARK(slot1) == 0 && MARK(slot2) == 0) ||
(MARK(slot4) == 2 && MARK(slot1) == 0 &&
MARK(slot2) == 2) || (MARK(slot4) == 2 &&
MARK(slot1) == 2 && MARK(slot2) == 0) ||
(MARK(slot4) == 2 && MARK(slot1) == 2 &&
MARK(slot2) == 2)) {</td></tr>
</table>

Table 2.5: Activity case probabilities for SAN example

Activity	*Case*	*Probability*
		return(ZERO); } } *else* { *return(ZERO);* }

third position in zero time if there is no job at the third position. When this happens, *case1* of the activity *I4*, which models this forward movement of a job, is assigned a probability of one, since jobs only become impatient upon completion of a service. Similarly, *in3*, *I3*, and *out3* shift jobs from the third to second position, while *in2* and *I2* shift jobs from the second to first position. Therefore, if there are zero customers in the queue, an entering customer at the fourth position will move to the first position in zero time. (See the appendix for complete illustration of gate functions and marking-dependent case probabilities.)

Given the SAN description of the system, we can define a reward structure corresponding to the performance measure of interest. In particular, let X be the number of jobs that depart from our example queue without obtaining service during some interval of time. Suppose we are interested in finding the mean, variance, and distribution of X. In terms of step-based reward structure, X can be formulated as follows:

$$R(r) = 0, \quad \forall r$$

$$C(\tau) = \begin{cases} 1 & \text{if } \tau \in \{\tau_1 = (\star, I4, c2), \tau_2 = (\star, I3, c2)\} \\ 0 & \text{otherwise.} \end{cases}$$

Since the jobs which become impatient can leave the system by choice of *case2* after completion of instantaneous activity *I3* or *I4*, to compute X, these events must be counted over the observation period. This can be achieved by collecting an impulse reward of one for each occurrence of these events. Note X cannot be defined in terms of rate rewards.

2.4 Well-Specified SANs

To solve for a desired performance variable, we need to transform a SAN and the associated step-based reward structure into a state-level reward model. The state-level reward model consists of a stochastic process and a reward structure defined on the process. The generation of a state-level reward model from a SAN and associated step-based reward structure requires translation of probabilistic behavior of the SAN, as defined by its execution rules, to the stochastic process level. To a large extent, this translation can be carried out in a straightforward manner by considering the probabilistic behavior of timed activities and case distributions. However, if multiple instantaneous activities are simultaneously enabled, the generation of the process will be complicated by the fact that these activities are not assigned priorities or probabilities. As a consequence, there is a concern whether the SAN specification is completely probabilistically specified, in the sense that it uniquely defines a stochastic process. In this section, we will discuss this issue and give conditions under which a SAN will be "well specified" with respect to a given step-based reward structure. These conditions will be such that they can be checked during the generation of the stochastic process from the SAN and the step-based reward structure.

Recall that the probabilistic behavior in SANs is specified by two types of probability distributions: 1) a case distribution assigned to cases of each timed or instantaneous activity and 2) an activity time distribution assigned to each timed activity. Case distributions quantify the nondeterminism regarding *case choice*, i.e., the choice of one of the cases of the just completed activity. The nondeterminism due to choice of timed activities is quantified by their activity time distributions.[1] However, uncertainty regarding which instantaneous activity, among several, completes is not quantified, since instantaneous activities complete in zero time. For convenience, we will refer to the activity choice among two or more simultaneously enabled instantaneous activities as a *non-quantified activity choice.*

Other extensions of SPNs such as GSPNs [9, 40] and SRNs [8] have circumvented the problem of non-quantified activity choices by introducing probabilities or priorities to simultaneously enabled instantaneous activities. From the perspective of the modeler, however, assigning probabilities or priorities is not always natural, since it may require specifying probabilities or priorities on events where the order does not really matter. We therefore take a different approach, where choices that matter to a modeler are specified with cases, and choices among simultaneously enabled instantaneous activities are checked to insure that they do not matter. When this is the case, the SAN with the step-based reward structure is "well specified."

Informally, a SAN with a step-based reward structure is well specified if, for every stable reachable marking $\mu \in SR(SAN, \mu_0)$ and for every timed activity a which may complete in μ:

1. the probability of reaching a particular next stable marking, given that a completes in μ, is independent of non-quantified activity choices, and

[1] Note that if multiple activities have discrete or mixed distributions, there is a possibility that they may complete at the same time. This ambiguity can either be resolved in the same manner as done here for instantaneous activities, or by probabilities or priorities, and hence will not be discussed further.

2. the probability of obtaining a certain impulse reward when entering a particular next stable marking, given that a completes in μ, is independent of non-quantified activity choices.

If these two conditions are met, the SAN is well specified. To investigate these conditions more formally, we first define two measures on each stable step. The first measure expresses the probability of the occurrence of a particular stable step given that a particular timed activity completes in a particular stable reachable marking. Specifically,

Definition 2.4.1: Let $s = <\mu_1, a_1, c_1><\mu_2, a_2, c_2>\ldots<\mu_n, a_n, c_n><\mu_{n+1}, -, ->$ be a stable step. Then $P\{s|\mu_1, a_1\}$, *the probability of s given completion of timed activity a_1 in μ_1* is

$$P\{s|\mu_1, a_1\} = p_{a_1}(c_1) \times p_{a_2}(c_2) \times \ldots \times p_{a_n}(c_n),$$

where $p_{a_i}(c_i)$, for $1 \leq i \leq n$, is the probability that case c_i is chosen when activity a_i completes.

The second measure expresses the impulse reward obtained by completion of a particular stable step and depends on the step-based reward structure defined earlier. Specifically,

Definition 2.4.2: $i(s)$, the *impulse reward assigned to a stable step s*, is the sum of the impulse rewards assigned to any partial step that appears in the stable step and the impulse reward assigned to the stable step itself.

Furthermore, we divide stable steps into sets $S(\mu, a)$, where each set contains stable steps which occur upon completion of a particular timed activity a in some particular stable marking μ. Formally,

$$S(\mu, a) \; = \; \left\{ s \;\middle|\; \begin{array}{l} s \text{ is a stable step where timed} \\ \text{activity } a \text{ completes in marking } \mu \end{array} \right\}.$$

According to the first condition in our informal definition, we need to check whether all non-quantified activity choices give the same probability of reaching a particular next stable marking for each set $S(\mu, a)$. (We assume that the SAN is stabilizing [2], and hence the set $S(\mu, a)$ is always finite.) To understand the non-quantified activity choices that may occur, note that each stable step in the set $S(\mu, a)$ differs from every other stable step in the set in selecting different activity and/or case choices. If all stable steps in the set $S(\mu, a)$ only differ in their case choice, no non-quantified activity choices need be made when a completes in μ. In this situation, it is obvious that the SAN is well specified with respect to the reward structure. On the other hand, if steps in the set $S(\mu, a)$ differ from each other in that different instantaneous activities complete in the same unstable marking, one or more non-quantified activity choices must be made when a completes in μ. To identify whether two stable steps s and s' in $S(\mu, a)$ make the same non-quantified activity choices, we define the following relation on $S(\mu, a)$:

$$R \; = \; \left\{ (s, s^{'}) \;\middle|\; \begin{array}{l} \text{for every configuration} < \mu, a, c > \\ \text{in } s \text{ and configuration} < \mu^{'}, a^{'}, c^{'} > \\ \text{in } s^{'} \text{ such that } \mu = \mu^{'},\, a = a^{'} \end{array} \right\}.$$

Two stable steps are thus related if, for every unstable marking they share in common, the same activity choice is made. R is a compatibility relation, i.e., it is reflexive and symmetric. Although a compatibility relation does not necessarily define a partition of a set, it does define a covering of a set, by the maximal compatibility classes of the relation. A subset $C(\mu, a) \subseteq S(\mu, a)$ is called a *maximal compatibility class* if every element of $C(\mu, a)$ and no element of $S(\mu, a) - C(\mu, a)$ is related to any element of $C(\mu, a)$.

If a non-quantified activity choice is made, steps that choose different instantaneous activities will belong to different compatibility classes. Therefore, for the first condition in our informal definition to be met, it is necessary that the probability of reaching each possible next stable marking be invariant across maximal compatibility classes. To check this, define $NS(\mu, a)$ as the set of next possible stable markings when a completes in μ.

Definition 2.4.3: Let $\mu \in SR(SAN, \mu_0)$. The set of *next stable markings* for the SAN in μ upon completion of the timed activity a is the set

$$NS(\mu, a) = \left\{ \mu' \;\middle|\; \begin{array}{l} \exists \text{ a stable step } s \text{ from } \mu \text{ to } \mu' \\ \text{such that } a \text{ is the activity of} \\ \text{the first configuration of } s \end{array} \right\}.$$

On the basis of next stable reachable markings, we then partition the stable steps within the compatibility class $C(\mu, a)$ into sets $C(\mu, a, \mu')$ where each set $C(\mu, a, \mu')$ contains the stable steps whose final marking is $\mu' \in NS(\mu, a)$. Formally,

$$C(\mu, a, \mu') = \left\{ s \;\middle|\; \begin{array}{l} \text{is a stable step from } \mu \text{ to} \\ \mu' \text{ in } C(\mu, a) \end{array} \right\}.$$

To verify that stable steps in different compatibility classes give the same probability of reaching a particular next stable marking, we compute $P_{C(\mu,a)}\{\mu' | \mu, a\}$. $P_{C(\mu,a)}\{\mu' | \mu, a\}$ is the probability of reaching a stable marking $\mu' \in NS(\mu, a)$ when the timed activity a

completes in μ and the set of non-quantified activity choices corresponding to compatibility set $C(\mu, a)$ is chosen. Formally,

$$P_{C(\mu,a)}\{\mu^{'}|\mu, a\} \;=\; \sum_{s \in C(\mu,a,\mu^{'})} P\{s|\mu, a\}. \tag{2.1}$$

Given this expression, the first condition for a SAN to be well specified holds when, for all $\mu^{'} \in NS(\mu, a)$, $P_{C(\mu,a)}\{\mu^{'}|\mu, a\}$ is the same for all maximal compatibility classes $C(\mu, a)$.

The second condition for a SAN to be well specified can be verified in a similar manner. To do this, we first define the set of impulse rewards $I(\mu, a, \mu^{'})$, where

$$I(\mu, a, \mu^{'}) \;=\; \left\{ i(s) \;\middle|\; \begin{array}{l} s \in S(\mu, a) \text{ and has next} \\ \text{stable marking } \mu^{'} \end{array} \right\}.$$

Furthermore, we partition the set $C(\mu, a, \mu^{'})$ into sets $C(\mu, a, \mu^{'}, i)$, where each set $C(\mu, a, \mu^{'}, i)$ contains the stable steps which have impulse reward equal to $i \in I(\mu, a, \mu^{'})$. Formally,

$$C(\mu, a, \mu^{'}, i) \;=\; \{s \mid s \in C(\mu, a, \mu^{'}) \text{ and } i(s) = i\}.$$

After forming sets $C(\mu, a, \mu^{'}, i)$, we compute $P_{C(\mu,a)}\{\mu^{'}, i \mid \mu, a\}$. For a maximal compatibility set $C(\mu, a)$, $P_{C(\mu,a)}\{\mu^{'}, i \mid \mu, a\}$ is the probability of reaching stable marking $\mu^{'}$ and obtaining impulse reward equal to i when the timed activity a completes in the marking μ. Specifically,

$$P_{C(\mu,a)}\{\mu^{'}, i \mid \mu, a\} \;=\; \sum_{s \in C(\mu,a,\mu^{'},i)} P\{s \mid \mu, a\}. \tag{2.2}$$

Furthermore, for each compatibility set $C(\mu, a)$, we define $P_{C(\mu,a)}\{i \mid \mu, a, \mu^{'}\}$ as the probability an impulse of i is obtained when stable marking $\mu^{'}$ is reached by completion of timed activity a in μ. $P_{C(\mu,a)}\{i \mid \mu, a, \mu^{'}\}$ can be expressed as:

$$P_{C(\mu,a)}\{i \mid \mu, a, \mu^{'}\} \;=\; \frac{P_{C(\mu,a)}\{\mu^{'}, i \mid \mu, a\}}{P_{C(\mu,a)}\{\mu^{'} \mid \mu, a\}}. \tag{2.3}$$

Finally, to fulfill the second condition for a SAN to be well specified, $P_{C(\mu,a)}\{i \mid \mu, a, \mu^{'}\}$ must be identical over all maximal compatibility classes $C(\mu, a)$. The second condition can thus be checked in a manner identical to that for the first condition, resulting in the following precise condition for a SAN to be well specified.

Definition 2.4.4: A SAN with a step-based reward structure is *well specified* if, for every $\mu \in SR(SAN, \mu_0)$, every timed activity a that is enabled in μ, and every $\mu^{'} \in NS(\mu, a)$:

1. $P_{C(\mu,a)}\{\mu^{'} \mid \mu, a\}$ is the same for all maximal compatibility classes $C(\mu, a) \subseteq S(\mu, a)$, and
2. $P_{C(\mu,a)}\{i \mid \mu, a, \mu^{'}\}$ is the same for all maximal compatibility classes $C(\mu, a) \subseteq S(\mu, a)$, for each $i \in I(\mu, a, \mu^{'})$.

When a SAN is well specified, we can thus drop the subscript $C(\mu, a)$ from $P_{C(\mu,a)}\{\mu^{'} \mid \mu, a\}$ and $P_{C(\mu,a)}\{i \mid \mu, a, \mu^{'}\}$, since these probabilities are invariant over different compatibility sets $C(\mu, a) \in S(\mu, a)$. Accordingly, $P\{\mu^{'} \mid \mu, a\}$ gives the probability of entering stable marking $\mu^{'}$ when a timed activity a completes in μ. Likewise, $P\{i \mid \mu, a, \mu^{'}\}$ gives the probability an impulse $i \in I(\mu, a, \mu^{'})$ is obtained when the stable marking $\mu^{'}$ is reached by completion of timed activity a in μ.

2.5 Example Revisited

To illustrate the computation of the sets, compatibility classes, and probabilities discussed in this section, we consider again the SAN example in Figure 2.1. Specifically, we consider the set $S((0,1,2,1,2), sv)$ which consists of the stable steps that may occur when the timed activity sv completes in stable reachable marking (0,1,2,1,2). In this expression, and the following, we write markings as a vector (*arrival*, *slot4*, *slot3*, *slot2*, *slot1*), where the number in a position indicates the number of tokens in the corresponding place. The set $S((0,1,2,1,2), sv)$ then consists of the following stable steps:

$$
\begin{aligned}
s_1 = \quad & <(0,1,2,1,2), sv, c_1><(0,1,0,1,0), I2, c_1><(0,1,0,0,1), I4, c_2> \\
& <(0,0,0,0,1), -, ->, \\
s_2 = \quad & <(0,1,2,1,2), sv, c_1><(0,1,0,1,0), I2, c_1><(0,1,0,0,1), I4, c_1> \\
& <(0,0,1,0,1), I3, c_2><(0,0,0,0,1), -, ->, \\
s_3 = \quad & <(0,1,2,1,2), sv, c_1><(0,1,0,1,0), I4, c_2><(0,0,0,1,0), I2, c_1> \\
& <(0,0,0,0,1), -, ->, \\
s_4 = \quad & <(0,1,2,1,2), sv, c_1><(0,1,0,1,0), I4, c_1><(0,0,1,1,0), I2, c_1> \\
& <(0,0,1,0,1), I3, c_2><(0,0,0,0,1), -, ->, \\
s_5 = \quad & <(0,1,2,1,2), sv, c_1><(0,1,0,1,0), I2, c_1><(0,1,0,0,1), I4, c_1> \\
& <(0,0,1,0,1), I3, c_1><(0,0,0,1,1), -, ->, \\
s_6 = \quad & <(0,1,2,1,2), sv, c_1><(0,1,0,1,0), I4, c_1><(0,0,1,1,0), I2, c_1> \\
& <(0,0,1,0,1), I3, c_1><(0,0,0,1,1), -, -> .
\end{aligned}
$$

Recall that cases c_1 and c_2 of activity *I4* are assigned probabilities depending on the number of services after which the job in place *slot4* will acquire the server. Similarly, cases c_1 and c_2

of activity *I3* are assigned probabilities depending on the number of services after which the job in place *slot3* will acquire the server. In stable steps s_1, s_2, s_3, and s_4, the probabilities assigned to cases of both activities *I3* and *I4* are identical, since there is always one job ahead of the job in *slot3* and *slot4*. Let p be the probability assigned to case c_1 of the activities *I3* and *I4*, and $(1-p)$ be the probability assigned to case c_2 of these activities. Then, in Definition 2.4.1, the summation is carried out over the elements

$$\begin{aligned}
P\{s_1 \mid (0,1,2,1,2), sv\} &= (1-p) \\
P\{s_2 \mid (0,1,2,1,2), sv\} &= p(1-p), \\
P\{s_3 \mid (0,1,2,1,2), sv\} &= (1-p), \\
P\{s_4 \mid (0,1,2,1,2), sv\} &= p(1-p), \\
P\{s_5 \mid (0,1,2,1,2), sv\} &= p^2, \\
P\{s_6 \mid (0,1,2,1,2), sv\} &= p^2.
\end{aligned}$$

Furthermore, recall that our performance measure is the number of jobs that depart from the queue without obtaining service. For this measure, according to Definition 2.4.2, $i(s_1) = i(s_2) = i(s_3) = i(s_4) = 1$ and $i(s_5) = (s_6) = 0$.

Then, according to the definition of the relation R and $S((0,1,2,1,2), sv)$, two sets of maximal compatibility classes, C_1 and C_2, result, where

$$C_1 = \{s_1,\ s_2,\ s_5\},\quad C_2 = \{s_3,\ s_4,\ s_6\}.$$

Furthermore, according to Definition 2.4.3, the set of next stable markings reachable upon completion of timed activity sv in marking (0,1,2,1,2) can be computed as

$$NS((0,1,2,1,2), sv) = \{(0,0,0,0,1), (0,0,0,1,1)\}.$$

On the basis of this set of next stable markings, we compute:

$$\begin{aligned}
C_1((0,1,2,1,2), sv, (0,0,0,0,1)) &= \{s_1, s_2\} \\
C_1((0,1,2,1,2), sv, (0,0,0,1,1)) &= \{s_5\} \\
C_2((0,1,2,1,2), sv, (0,0,0,0,1)) &= \{s_3, s_4\} \\
C_2((0,1,2,1,2), sv, (0,0,0,1,1)) &= \{s_6\}.
\end{aligned}$$

Furthermore, using Equation 2.1, the following probabilities are computed:

$$\begin{aligned}
P_{C_1}\{(0,0,0,0,1) \mid (0,1,2,1,2), sv\} &= (1-p^2), \\
P_{C_1}\{(0,0,0,1,1) \mid (0,1,2,1,2), sv\} &= p^2, \\
P_{C_2}\{(0,0,0,0,1) \mid (0,1,2,1,2), sv\} &= (1-p^2), \\
P_{C_2}\{(0,0,0,1,1) \mid (0,1,2,1,2), sv\} &= p^2.
\end{aligned}$$

These probabilities imply that the first condition required for a SAN to be well specified is true when activity sv completes in marking $(0,1,2,1,2)$.

Similarly, from the knowledge of the impulse reward associated with each stable step, we compute the following sets:

$$\begin{aligned}
I((0,1,2,1,2), sv, (0,0,0,0,1)) &= \{1\}, \\
I((0,1,2,1,2), sv, (0,0,0,1,1)) &= \{0\},
\end{aligned}$$

On the basis of these sets of impulse reward, we then compute:

$$\begin{aligned}
C_1((0,1,2,1,2), sv, (0,0,0,0,1), 1) &= \{s_1, s_2\}, \\
C_1((0,1,2,1,2), sv, (0,0,0,1,1), 0) &= \{s_5\}, \\
C_2((0,1,2,1,2), sv, (0,0,0,0,1), 1) &= \{s_3, s_4\}, \\
C_2((0,1,2,1,2), sv, (0,0,0,1,1), 0) &= \{s_6\}.
\end{aligned}$$

Furthermore, using Equation 2.2, we can compute the following probabilities:

$$
\begin{aligned}
P_{C_1}\{1,(0,0,0,0,1) \mid (0,1,2,1,2), sv\} &= (1-p^2), \\
P_{C_1}\{0,(0,0,0,1,1) \mid (0,1,2,1,2), sv\} &= p^2, \\
P_{C_2}\{1,(0,0,0,0,1) \mid (0,1,2,1,2), sv\} &= (1-p^2), \\
P_{C_2}\{0,(0,0,0,1,1) \mid (0,1,2,1,2), sv\} &= p^2.
\end{aligned}
$$

Finally, using Equation 2.3, we compute:

$$
\begin{aligned}
P_{C_1}\{1 \mid (0,0,0,0,1) \mid (0,1,2,1,2), sv\} &= 1, \\
P_{C_1}\{0 \mid (0,0,0,1,1) \mid (0,1,2,1,2), sv\} &= 1, \\
P_{C_2}\{1 \mid (0,0,0,0,1) \mid (0,1,2,1,2), sv\} &= 1, \\
P_{C_2}\{0 \mid (0,0,0,1,1) \mid (0,1,2,1,2), sv\} &= 1.
\end{aligned}
$$

Thus, the second required condition in Definition 2.4.4 holds for this SAN and reward structure when activity sv completes in marking $(0,1,2,1,2)$. Identical computations could be made to check the other reachable stable markings, showing that the SAN is indeed well specified with respect to this reward structure.

In the next section, we present an algorithm to generate stable reachable markings of a SAN which embodies the well-specified check.

2.6 Generation of the State-Level Reward Model

The generation of a stochastic process underlying a SAN and its associated net level reward structure can be done in an algorithmic fashion. We will present in Section 2.6.1 an algorithm which generates the stable reachable markings and determines whether the SAN is well specified, using the approach described in the previous sections. If the SAN is well specified, it generates the sets $NS(\mu, a)$ and $I(\mu, a, \mu')$ and the probabilities $P\{\mu'|\mu, c\}$ and

$P\{i|\mu, a, \mu'\}$, where $\mu' \in NS(\mu, a)$ and $i \in I(\mu, a, \mu')$. From this information, the state-level stochastic process and the associated state-level reward structure can be obtained. In Section 2.6.2, we will discuss in detail the resulting state-level representation if states are chosen to correspond to stable reachable markings.

2.6.1 Algorithm for Generating Stable Reachable Markings

SANs require that the initial marking of a SAN must be stable. Given a SAN and the initial marking μ_0, for each timed activity enabled in μ_0, the set of stable steps $S(\mu_0, a_i)$ is computed. After computation of a set $S(\mu_0, a_i)$, the conditions for a well-specified SAN are verified. As part of the well-specified check, first sets $NS(\mu_0, a_i)$ and $I(\mu, a, \mu^{'})$ and probabilities $P\{\mu^{'} \mid \mu_0, a_i\}$ and $P\{i \mid \mu_0, a_i, \mu^{'}\}$ are computed, where $\mu^{'} \in NS(\mu_0, a_i)$ and $i \in I(\mu, a, \mu^{'})$. This procedure is then repeated for every stable reachable marking until no more new stable reachable markings are generated. Using this approach, we present an algorithm which, given a SAN and its initial marking, generates all the stable reachable markings if the SAN is well specified, otherwise it exits with the signal that the SAN is not well specified.

The algorithm is presented in a modular form where the main algorithm, for a stable reachable marking and an activity which may complete in it, invokes a second algorithm which performs the well-specified check.

Algorithm 2.6.1 *(Generates the set, SR, of stable reachable markings and, for every $\mu \in SR(SAN, \mu_0)$ and every timed activity a which may complete in μ, computes $NS(\mu, a)$, $I(\mu, a, \mu^{'})$, $P\{\mu^{'} \mid \mu, a\}$, and $P\{i \mid \mu, a, \mu^{'}\}$ where $\mu^{'} \in NS(\mu, a)$ and $i \in I(\mu, a, \mu^{'})$.)*

$SR \ = \ \emptyset$. *(set of stable reachable markings)*

$U \ = \ \{\mu_0\}$. *(set of unexplored reachable markings)*

While $U \ \neq \ \emptyset$.

  For some $\mu \ \in \ U$:

    $SR \ = \ SR \cup \{\mu\}$.

    Compute the set of enabled timed activities

    E_T *in* μ.

    For each timed activity $a \in E_T$:

      By invoking Algorithm 2, determine

      if the conditions for the SAN to be well-

      specified hold in a marking μ *when* a

      completes.

      If the conditions do not hold, signal that

      the SAN is not well specified and exit

      the algorithm.

      Else, return $NS(\mu, a)$, $I(\mu, a, \mu^{'})$,

      $P\{\mu^{'} \mid \mu, a\}$, *and* $P\{i \mid \mu, a, \mu^{'}\}$.

      $U \ = \ (U \cup NS(\mu, a)) - SR$.

While end.

Algorithm 2.6.2 *(Determines if conditions for a SAN to be well specified hold in a marking* μ *when activity* a *completes. If so, return* $NS(\mu,a)$, $I(\mu,a,\mu^{'})$, $P\{\mu^{'} \mid \mu,a\}$, *and* $P\{i \mid \mu,a,\mu^{'}\}$ *where* $\mu^{'} \in NS(\mu,a)$ *and* $i \in I(\mu,a,\mu^{'})$.*)*

Compute $S(\mu,a)$, *the set of all stable steps that can*
result by completion of timed activity a *in stable*
reachable marking μ.

Compute $NS(\mu,a)$ *using* $S(\mu,a)$.

Construct the maximal compatibility sets
$C(\mu,s) \subseteq S(\mu,a)$.

For each maximal compatibility set $C(\mu,a)$:

  Compute $P_{C(\mu,a)}\{\mu^{'} \mid \mu,a\}$ *and*
  $P_{C(\mu,a)}\{i \mid \mu,a,\mu^{'}\}$.

Next $C(\mu,a)$.

If $P_{C(\mu,a)}\{\mu^{'} \mid \mu,a\}$ *and* $P_{C(\mu,a)}\{i \mid \mu,a,\mu^{'}\}$ *are*
identical for all maximal compatibility sets:

  Return $NS(\mu,a)$, $I(\mu,a,\mu^{'})$, $P\{\mu^{'} \mid \mu,a\}$,
  and $P\{i \mid \mu,a,\mu^{'}\}$.

Else:

Return that the SAN is not well specified.

2.6.2 State-Level Representation

The final step in obtaining a state-level representation of a SAN is to construct a state-level reward structure and corresponding stochastic process. Several types of stochastic processes can result from the SAN specification. Ciardo *et al.* [17], for example, distinguish Markov SPNs, semi-Markov SPNs, semi-regenerative SPNs, and generalized semi-Markov SPNs, depending on the type of underlying stochastic process which represents the marking behavior (Markov, semi-Markov, semi-regenerative, and generalized semi-Markov processes, respectively). The above-mentioned stochastic processes are of particular interest since analytic solutions have been derived to solve for the state space occupancy distribution and for reward measures.

In this section, we discuss how a state-level reward structure can be obtained for stochastic processes as mentioned above. In particular, we will assume that the states in the stochastic process correspond to stable reachable markings. Furthermore, we will discuss how the general state-level representation can be simplified if the stochastic process is Markov.

If rate rewards are considered, rates must be associated with certain states (stable markings) in the generated process. Since more than one rate reward can be assigned to a stable marking (for instance through several partial markings), the total rate reward associated with a state will be the sum of all the rate rewards assigned to the corresponding stable marking. As a result, we then obtain the following rate reward vector $\mathbf{r}$:

$$\mathbf{r} \;=\; [r_1, r_2, \ldots, r_n],$$

where n is the number of states of the stochastic process.

The state-space-level representation for impulse rewards is more complicated because transitions between states must be considered. From the algorithm that generates the stable reachable markings, the following results are available: the set of impulses $I(\mu, a, \mu')$, and the probabilities $P\{\mu'|\mu, a\}$ and $P\{i|\mu, a, \mu'\}$, for all $\mu \in SR$, $\mu' \in NS(\mu, a)$, $i \in I(\mu, a, \mu')$, and all activities a enabled in μ. Since completion of different timed activities (possibly with different associated impulse rewards) can result in a transition between the same states, we distinguish for every transition the activity whose completion causes the transition to occur. Furthermore, since any activity completion can result in various impulse rewards (depending on the instantaneous activities "following" the activity), a probability distribution over all possible impulse rewards is assigned to each activity.

To describe this more formally, let v and w be states in the stochastic process and μ_v and μ_w denote the corresponding stable markings. Then, for all activities a that can result in a transition from state v to state w, $i^a_{v,w}$ is defined as a vector with probabilities

$$P\{\mu_w, i|\mu_v, a\} = P\{i|\mu_v, a, \mu_w\} \cdot P\{\mu_w|\mu_v, a\}.$$

So, $i^a_{v,w}$ has $|I(\mu_v, a, \mu_w)|$ elements. Furthermore, let $i_{v,w}$ be a vector consisting of the vectors $i^a_{v,w}$, i.e.,

$$i_{v,w} = (i^{a_1}_{v,w}, \cdots, i^{a_n}_{v,w}),$$

where $a_1, \cdots, a_n$, are all activities that result in a transition from state v to w (so, $i_{v,w}$ has dimension $\sum_{k=1}^{n} |I(\mu_v, a_k, \mu_w)|$). Then, the state-space-level impulse reward structure is represented by the following matrix:

$$\underline{\mathbf{I}} = \begin{bmatrix} i_{1,1} & i_{1,2} & \dots & i_{1,n} \\ i_{2,1} & i_{2,2} & \dots & i_{2,n} \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ i_{n,1} & i_{n,2} & \dots & i_{n,n} \end{bmatrix}.$$

To close this section, we illustrate how the state-space-level definition simplifies if the underlying stochastic process is a continuous-time Markov process. In this case, the stochastic process can be represented by a generator matrix containing the rate of the transitions of the exponentially distributed activities. Let $\lambda_{a_k}(\mu_v), k = 1, \cdots, n$, be the rate of the exponentially distributed activities $a_1, \cdots, a_n$, which result in a transition from state v to w. Then, the (v, w)-th entry in the generator matrix is

$$\lambda_{v,w} = \sum_{k=1}^{n} \lambda_{a_k}(\mu_v) \cdot P\{\mu_w | \mu_v, a_k\}.$$

Furthermore, the distribution over impulse rewards, represented by the vector $i_{v,w}$, can be computed by

$$P\{i | \mu_v, \mu_w\} = \sum_{k=1}^{n} P\{i | \mu_v, a_k, \mu_w\} \cdot P\{a_k | \mu_v, \mu_w\},$$

for all $i \in I(\mu_v, a_k, \mu_w)$, $k = 1, \cdots, n$. In this way, the probability of occurrence of identical impulse rewards is summed for the activities $a_k, k = 1, \cdots, n$. The probability $P\{a_k | \mu_v, \mu_w\}$ is available from the algorithm for generating stable marking, via the relation

$$P\{a_k | \mu_v, \mu_w\} = \frac{\lambda_{a_k}(\mu_v) \cdot P\{\mu_w | \mu_v, a_k\}}{\lambda_{v,w}}.$$

So, we see that if the generator matrix and the probabilities $P\{i|\mu_v, \mu_w\}$ are available, a Markov SAN with step-based reward structure is completely defined on the state level.

CHAPTER 3

Computation of State-Occupancy Probability Vector

Chapter 2 defines a step-based reward structure which enables one to specify performance/dependability variables at the stochastic activity network level. As discussed in Chapter 1, the resulting reward (specified) model is then solved to either obtain a steady-state or a transient (instant-of-time or interval-of-time) solution. This chapter presents a method to solve the specified model to compute instant-of-time measures.

In general, instant-of-time measures can be determined by computing the state-occupancy probability vector $\pi(t)$. When the specified model is a reward model, the occupancy probabilities of states with identical rate rewards are added together to compute probabilities of obtaining different rate rewards. Probabilities of different rate rewards can then be used to compute the expected value and probability distribution of an instant-of-time variable.

As discussed in Chapter 1, many algorithms have been proposed to compute the state-occupancy probability vector of continuous-time time-homogeneous Markov processes. The most popular ones are based either on ordinary differential equation methods or uniformization. Algorithms based on ordinary differential equation (ODE) methods are limited in their application, since their performance is very sensitive to factors such as the desired accuracy of the solution, stiffness of the system, value of the time point, etc. Algorithms based on uniformization often perform better than algorithms based on ODE methods but their performance degrade rapidly with the increase in the value of the product $\lambda \times t$, where λ is the maximum total departure rate from states in the model and t is the observed time point. This rapid degradation in performance of algorithms is experienced because uniformization computes the state-occupancy probabilities of a continuous-time Markov process in terms of the state occupancy-probabilities of a discrete-time Markov chain. The discrete-time Markov chain is generated from the continuous-time Markov process such that the transitions of the discrete-time Markov chain occur according to a Poisson process whose rate is equal to the maximum of the sum of rates out of each state.

In this chapter, we propose a method based on exploring paths of the continuous-time Markov process in a depth-first search manner to compute the state-occupancy probability vector without constructing a corresponding DTMC. These paths can be generated directly from a high-level specification language (such as SANs) without first constructing the state-level reward model. Unlike uniformization, the algorithm works directly on the continuous-time Markov process rather than a discrete-time Markov chain. The new method is thus less sensitive to the stiffness of the system than uniformization.

The remainder of the chapter is organized as follows. Section 3.1 proposes a formalism based on exploring paths of a continuous-time Markov process to compute an instant-of-time measure. Section 3.2 develops a methodology based on a renewal argument to calculate the probability of a path. Section 3.3 uses the methodology developed in Section 3.2 to construct an overall algorithm to compute the instant-of-time measures. Section 3.4 illustrates examples to show the usefulness of the algorithm.

3.1 Path-Based Formalism

Consider a continuous-time time-homogeneous Markov process $X = \{X_t : t \geq 0\}$ with M states. Assume that an instant-of-time reward variable $R(t)$ is defined by assigning k distinct rate rewards $r_1 > r_2 > \ldots > r_k$, where $1 \leq k \leq M$, to the states of the process X, where $s(r_i)$ is the set of states with rate reward r_i. Then, the probability of being in a state with reward r_i at time t, $p_{s(r_i)}(t)$, can be computed as

$$p_{s(r_i)}(t) \;=\; \sum_{a \in s(r_i)} \pi_a(t),$$

where $\pi_a(t)$ is the occupancy probability of state a at time t. Given $p_{s(r_i)}(t)$, the expected value and probability distribution of instant-of-time variable $R(t)$ can be expressed, respectively, as

$$E[R(t)] \;=\; \sum_{1 \leq i \leq k} p_{s(r_i)}(t) \times r_i, \tag{3.1}$$

and

$$F_{R(t)}(r) \;=\; \sum_{j \leq i \leq k} p_{s(r_i)}(t), \quad \text{where } r_j \text{ is the greatest rate reward less than or equal to } r. \tag{3.2}$$

The rest of this section develops a formalism to compute state-occupancy probabilities by enumerating paths of the stochastic process.

A path in a stochastic process is a possible sequence of states that the stochastic process can traverse. The *probability of occurrence of a path in an interval* $(0, t)$ is the probability of traversing the sequence in time t. The sum of probabilities of all the paths in an interval $(0, t)$ which have a particular state as their final state gives the state-occupancy probability of that state at time t. Accordingly, if a is a state in the process and P_a is the set of all paths with final state a, then the state-occupancy probability of state a at time t can be expressed as

$$\pi_a(t) \;=\; \sum_{p \in P_a} P_t(p), \tag{3.3}$$

where $P_t(p)$ is the probability of occurrence of path p in time $(0, t)$.

For acyclic finite-state processes, P_a is finite and therefore, the exact state-occupancy probability for state a can be computed directly using Equation 3.3. However, for cyclic processes, set P_a may be infinite. In this case, a finite number of highly probable paths are considered to compute an approximate value of the state-occupancy probability. Let $P_a^{'}$ be a finite set of such paths. Then an approximate value of the occupancy probability of state a, $\pi_a^{'}(t)$, can be computed as

$$\pi_a^{'}(t) \;=\; \sum_{p \in P_a^{'}} P_t(p).$$

Consequently, the sum of these approximate occupancy probabilities gives an approximate value of $p_{s(r_i)}(t)$, i.e.

$$p_{s(r_i)}^{'}(t) \;=\; \sum_{a \in s(r_i)} \pi_a^{'}(t),$$

Approximate values are only meaningful if we can compute reasonable bounds on the error induced by the approximation. Fortunately, it is straight forward to compute the exact error that is induced in the sum of the state-occupancy probabilities by considering a finite number of paths in a cyclic Markov process, since the sum of probabilities of all the paths of a stochastic process in time $(0, t)$ is one. Let $p_{error}(t)$ be the error induced by considering a finite set of paths $P^{'}$. Formally, then

$$p_{error}(t) \;=\; 1 - \sum_{p \in P^{'}} P_t(p). \tag{3.4}$$

Given $p_{error}(t)$, upper and lower bounds on the expected value of an instant-of-time variable can be computed by assuming that all the unconsidered paths $(P - P^{'})$ end up in states with the highest and lowest rate rewards, respectively. Accordingly,

$$E[R(t)]_{lower} \;=\; \sum_{1\leq i\leq k} p^{'}_{s(r_i)}(t) \times r_i \;+\; p_{error}(t) \times r_k, \tag{3.5}$$

and

$$E[R(t)]_{upper} \;=\; \sum_{1\leq i\leq k} p^{'}_{s(r_i)}(t) \times r_i \;+\; p_{error}(t) \times r_1. \tag{3.6}$$

Moreover $p_{error}(t)$ gives an upper bound on the distribution of the instant-of-time variable,

$$F_{R(t)_{upper}}(r) \;=\; \sum_{j\leq i\leq k} p^{'}_{s(r_i)}(t) + p_{error}(t). \tag{3.7}$$

Note that the probabilities of paths in the set $P^{'}$ gives a lower bount on the distribution,

$$F_{R(t)_{lower}}(r) \;=\; \sum_{j\leq i\leq k} p^{'}_{s(r_i)}(t). \tag{3.8}$$

Selection of finitely many paths from the infinite paths of a cyclic Markov process is achieved by selective truncation of paths. In our approach, a path is truncated if the sum of probabilities of all the paths that can be generated from it by further transitions of the Markov process is smaller than some weight w. For discrete-time Markov process, this criteria is very simple to check because the probability of a path also gives a bound on sum of probabilities of all the paths that can be generated from it. However, it is not true for the continuous-time Markov processes which we consider in our formalism. To compute this sum for continuous-time Markov processes, we recompute the probability of the path while assuming that the last state of the path is an absorbing state. This value minus the original probability of the path (without assuming that the last state is absorbing) gives the probability of all the paths that could be generated by transitions of the continuous-time Markov process.

The discarding criteria gives us a method to avoid exploring paths with low probabilities. As discussed earlier, we know how to compute the bound on the error induced by unexplored (discarded) low probability paths. Additionally, to evaluate an instant-of-time measure, we need an algorithm to compute the probability of a path in a continuous-time Markov process. The next section derives such an algorithm.

3.2 Calculation of Path Probabilities

Consider a continuous-time Markov process $\{X_t : t \geq 0\}$ with generator matrix $Q = [q_{i,j}]$, state-space M, and initial state s_0. Let $s_0, s_1, \ldots, s_n$, $n \geq 0$, be a path in the process. For $n = 0$, the path only consists of the initial state and its probability of occurence in interval $(0, t)$ can be obtained by simply computing $e^{-q_{s_0} t}$, where $q_{s_0} = |q_{s_0,s_0}|$. For $n \geq 1$, the probability of occurence of the path in an interval $(0, t)$ can be expressed, by using a renewal argument, as follows:

$$P_t\{s_0, s_1, \ldots, s_n\} = \int_0^t \left[P_{t_n}\{s_0, s_1, \ldots, s_{n-1}\} \times q_{s_{n-1},s_n} \times e^{\int_{t_n}^t -q_{s_n} du} \right] dt_n,$$

where $0 < t_n < t$ is the point in time when a transition is made from state s_{n-1} to state s_n, and $q_{s_n} = |q_{s_n,s_n}|$.

Let t_m, $1 \leq m \leq n$, be the point in time when the process makes a transition from state s_{m-1} to state s_m. (Note that $0 < t_1 < t_2 < \ldots < t_n < t$). Then by using the renewal argument repeatedly, we can express the probability of path $s_0, s_1, \ldots, s_n$ as

$$\begin{aligned} P_t\{s_0, s_1, \ldots, s_n\} &= q_{s_0,s_1} \times q_{s_1,s_2} \times \ldots \times q_{s_{n-1},s_{n-2}} \times \int_0^t \int_0^{t_n} \cdots \int_0^{t_2} e^{-q_{s_0} t_1} e^{-q_{s_1}(t_2 - t_1)} \\ &\quad e^{-q_{s_2}(t_3 - t_2)} \cdots e^{-q_{s_n}(t - t_n)}\, d_{t_1}\, d_{t_2} \cdots d_{t_n}. \end{aligned}$$

Note that

$$G(t) \;=\; \int_0^t \int_0^{t_n} \cdots \int_0^{t_2} e^{-q_{s_0}t_1} e^{-q_{s_1}(t_2-t_1)} e^{-q_{s_2}(t_3-t_2)} \cdots e^{-q_{s_n}(t-t_n)}\; d_{t_1}\; d_{t_2} \cdots d_{t_n}$$

is a convolution of $n+1$ functions. In the Laplace domain (z), this convolution can be expressed as

$$G(z) \;=\; \frac{1}{(z+q_{s_0})(z+q_{s_1})(z+q_{s_2})\cdots(z+q_{s_n})}.$$

In the Laplace domain, then, $P_t\{s_0, s_1, \ldots, s_n\}$ can be expressed as

$$P_z\{s_0, s_1, \ldots, s_n\} \;=\; q_{s_0,s_1} \times q_{s_1,s_2} \times \ldots \times q_{s_{n-1},s_{n-2}} \times G(z).$$

Suppose there are a, where $1 \le a \le n+1$, distinct rates among q_{s_0}, q_{s_1}, q_{s_2}, $\cdots$, $q_{s_{n-1}}$, q_{s_n}. Assume that these distinct rate rewards are denoted by λ_b where $1 \le b \le a$. Further, assume that k_b, $1 \le b \le a$, is the number of times rate λ_b appears in $G(z)$. This implies that

$$G(z) \;=\; \frac{1}{(z+\lambda_1)^{k_1}(z+\lambda_2)^{k_2}\cdots(z+\lambda_a)^{k_a}}, \quad \text{where} \quad \sum_{b=1}^{a} k_b = n+1.$$

By partial fraction expansion

$$G(z) \;=\; \sum_{b=1}^{a}\sum_{m=1}^{k_b} \frac{C_{b,m}}{(z+\lambda_b)^m}, \quad \text{where} \tag{3.9}$$

$$C_{b,m} \;=\; \frac{1}{(k_b-m)!} \lim_{s\to(-\lambda_b)} \frac{d}{dz}^{(k_b-m)} \left[(z+\lambda_b)^{k_b} G(z)\right],$$

for $b = 1, 2, \cdots, a$ and $m = 1, 2, \cdots, k_b$. Using Equation (3.9), we can write

$$P_z\{s_0, s_1, \ldots, s_n\} \;=\; q_{s_0,s_1} \times q_{s_1,s_2} \times \ldots \times q_{s_{n-1},s_n} \times \sum_{b=1}^{a}\sum_{m=1}^{k_b} \frac{C_{b,m}}{(z+\lambda_b)^m}. \tag{3.10}$$

To compute Equation 3.10, our strategy is to compute the partial fraction coefficients in the Laplace domain. If we are able to compute them in the Laplace domain, we can then take the inverse Laplace transform to obtain

$$P_t\{s_0, s_1, \ldots, s_n\} \;=\; Q_{s_0,s_1,\ldots,s_n} \times \sum_{b=1}^{a} \sum_{m=1}^{k_b} \frac{(\lambda_b t)^{m-1} e^{-\lambda_b t}}{(m-1)!} \times \frac{C_{b,m}}{{\lambda_b}^{m-1}}, \tag{3.11}$$

where $Q_{s_0,s_1,\ldots,s_n} = q_{s_0,s_1} \times q_{s_1,s_2} \times \ldots \times q_{s_{n-1},s_n}$. Equation 3.11 can then be rearranged as

$$P_t\{s_0, s_1, \ldots, s_n\} \;=\; \sum_{b=1}^{a} \sum_{m=1}^{k_b} \frac{(\lambda_b t)^{m-1} e^{-\lambda_b t}}{(m-1)!} \times C_{b,m} \times \frac{Q_{s_0,s_1,\ldots,s_n}}{{\lambda_b}^{m-1}}. \tag{3.12}$$

Equation 3.12 is susceptible to numerical errors due to the computation of constant coefficients $C_{b,m}$. In particular, computation of $C_{b,m}$ requires the computation of higher derivatives of the function

$$G_b(z) \;\equiv\; (z+\lambda_b)^{k_b} G(z) \;=\; \prod_{r=1,\ r\neq b}^{a} \frac{1}{(z+\lambda_r)^{k_r}},$$

since

$$C_{b,m} \;=\; \frac{1}{(k_b - m)!} \lim_{z\to(-\lambda_b)} \frac{d}{dz}^{(k_b-m)} \left[(z+\lambda_b)^{k_b} G(z)\right]. \tag{3.13}$$

In the remainder of this section, we derive a numerically stable algorithm to compute $C_{b,m}$. In computing each path occurence probability, the algorithm allows us to estimate the amount of required precision for the desired accuracy of the answer, which in turn enables us to selectively use extended precision whenever the required accuracy becomes greater than the machine accuracy.

3.2.1 Computation of $C_{b,m}$

To compute $C_{l,m}$, we use Bell polynomials, [41, 42], the higher derivatives of a composition of two functions. In particular, let $h = f \circ g$ be the composition of f with g, that is, $h(t) = f(g(t))$. Denote n-fold application of an operator d/dt by $d^{(n)}/dt^{(n)}$. Furthermore, denote $h_n = d^{(n)}h/dt^{(n)}$, $f_n = d^{(n)}f(g)/dg^{(n)}$, and $g_n = d^{(n)}g/dt^{(n)}$. Assume that all the higher-order derivatives of functions f, g, and h exist. Then the Bell polynomials can be expressed as follows:

$$
\begin{aligned}
h_1 &= f_1 g_1 \\
h_2 &= f_2 g_1^2 + f_1 g_2 \\
h_3 &= f_3 g_1^3 + f_2(3 g_1 g_2) + f_1 g_3 \\
&\vdots
\end{aligned}
$$

In general, $h_q = \sum_{p=1}^{q} f_p \alpha_{p,q}$, where the α_{pq}'s are polynomials in the g_i's that do not depend upon the choice of f. According to the Faa DiBurno's formula, [41], the α_{pq} can be explicitly represented as

$$\alpha_{p,q} = \sum_{\substack{n_i \geq 0 \\ \sum_{i=1}^{q} n_i = p \\ \sum_{i=1}^{q} i n_i = q}} \frac{q!}{(1!)^{n_1}(2!)^{n_2}\ldots(q!)^{n_q}(n_1!)(n_2!)\ldots(n_q!)} g_1^{n_1} g_2^{n_2} \ldots g_q^{n_q}. \tag{3.14}$$

Note that the summation is over all n_i such that $n_i \geq 0$, $\sum_{i=1}^{q} n_i = p$, and $\sum_{i=1}^{q} i n_i = q$. The computation of $\alpha_{p,q}$ for large values of q thus becomes prohibitive. This, in turn, makes the computation of higher derivatives of a composition of two functions impractical. However, if the function $h = f(g)$ is defined such that all its derivatives with respect to g are identical, then an efficient recursive expression can be written to compute the higher derivatives. The theorem gives us one such efficient recursion.

Lemma 3.2.1 *Let $h = f(g)$ be a function such that $h = f_1 = f_2 = \ldots$. Then h_q, for all $q \geq 1$, can be computed as*

$$h_q = \sum_{i=1}^{q-1} \begin{pmatrix} q-1 \\ q-i \end{pmatrix} g_i h_{q-i} + g_q h \ .$$

Proof: *Basis Step.* Let $q = 1$. Then the sum on the right is hg_1. From the definition of h_i, $h_1 = hg_1$.

Induction Hypothesis. Assume that for some $n \geq 1$,

$$h_n = \sum_{i=1}^{n} \begin{pmatrix} n-1 \\ n-i \end{pmatrix} g_i h_{n-i} + g_n h \ .$$

Induction Step. Note $h_{n+1} = d^{(n+1)}h/dt^{(n+1)} = d(h_n)/dt$. Accordingly,

$$\begin{aligned}
h_{n+1} &= \frac{d}{dt}\left[h_n = \sum_{i=1}^{n} \begin{pmatrix} n-1 \\ n-i \end{pmatrix} g_i h_{n-i} + g_n h \right] \\
&= \sum_{i=1}^{n} \begin{pmatrix} n-1 \\ n-i \end{pmatrix} \frac{d}{dt}\left[g_i h_{n-1}\right] + \frac{d}{dt}\left[hg_n\right] \\
&= \begin{pmatrix} n-1 \\ n-1 \end{pmatrix} \frac{d}{dt}\left[g_1 h_{n-1}\right] + \begin{pmatrix} n-1 \\ n-2 \end{pmatrix} \frac{d}{dt}\left[g_2 h_{n-2}\right] + \cdots + \\
&\quad \begin{pmatrix} n-1 \\ 2 \end{pmatrix} \frac{d}{dt}\left[g_{n-2} h_2\right] + \begin{pmatrix} n-1 \\ 1 \end{pmatrix} \frac{d}{dt}\left[g_{n-1} h_1\right] + g_n \frac{d}{dt} h + g_{n+1} h \\
&= \frac{d}{dt}\left[g_1 h_{n-1}\right] + (n-1)\frac{d}{dt}\left[g_2 h_{n-2}\right] + \cdots + \frac{(n-1)(n-2)}{2}\frac{d}{dt}\left[g_{n-2} h_2\right] \\
&\quad + (n-1)\frac{d}{dt}\left[g_{n-1} h_1\right] + g_n h_1 + g_{n+1} h \\
&= g_1 h_n + g_2 h_{n-1} + (n-1) g_2 h_{n-1} + (n-1) g_3 h_{n-2} + \cdots +
\end{aligned}$$

$$
\begin{aligned}
& \frac{(n-1)(n-2)}{2} g_{n-2} h_3 + \frac{(n-1)(n-2)}{2} g_{n-1} h_2 + \\
& (n-1) g_{n-1} h_2 + (n) g_n h_1 + h g_{n+1} \\
= \; & g_1 h_n + (n) g_2 h_{n-1} + \cdots + \left[\frac{(n-1)(n-2)}{2} + (n-1) \right] g_{n-1} h_2 \\
& (n) g_n h_1 + g_{n+1} h \\
= \; & \begin{pmatrix} n \\ n \end{pmatrix} g_1 h_n + \begin{pmatrix} n \\ n-1 \end{pmatrix} g_2 h_{n-1} + \cdots + \begin{pmatrix} n \\ 2 \end{pmatrix} g_{n-1} h_2 + \\
& \begin{pmatrix} n \\ 1 \end{pmatrix} g_n h_1 + g_{n+1} h \\
= \; & \sum_{i=1}^{n+1} \begin{pmatrix} n \\ n+1-i \end{pmatrix} g_i h_{n+1-i} + g_{n+1} h \; . \; \square
\end{aligned}
$$

We can use Lemma 3.2.1 to efficiently compute the higher derivatives of $G_b(-\lambda_b)$. To do this, we define $f(z) = \exp(z)$, $h(z) = f(g(z))$, and

$$
g(z) \;=\; -\sum_{r=1,\; r\neq b}^{a} k_r \ln(z + \lambda_r).
$$

Note that $h(z) = \exp(g(z)) = G_b(z)$. Furthermore, observe that

$$
\frac{d^{(p)}}{dg} G_b(g(z)) \;=\; \exp(g(z)) \;=\; G_b(-\lambda_b) \;=\; \prod_{r=1,\; r\neq b}^{a} (\lambda_r - \lambda_b)^{-k_r}, \tag{3.15}
$$

for all $p \geq 1$. Using Lemma 3.2.1, $G^{(k_b-m)}(-\lambda_b) = h_{k_b-m}$ can thus be expressed as

$$
G_b^{(k_b-m)}(-\lambda_b) \;=\; \sum_{i=1}^{k_b-m-1} \begin{pmatrix} k_b - m - 1 \\ k_b - m - i \end{pmatrix} g_i G_b^{(k_b-m-i)}(-\lambda_b) + g_{k_b-m} G_b(-\lambda_b), \tag{3.16}
$$

where

$$g_p = (-1)^p (p-1)! \sum_{\substack{r=1 \\ r \neq b}}^{a} \frac{k_r}{(\lambda_r - \lambda_b)^p} \quad \text{for } p \geq 1.$$

By looking at Equation (3.16) and using an inductive argument, it can be realized that

$$G_b^{(k_b-m)}(-\lambda_b) = G_b(-\lambda_b) \times \tilde{G}_b^{(k_b-m)}(-\lambda_b), \text{ where} \tag{3.17}$$

$$\tilde{G}_b^{(k_b-m)}(-\lambda_b) = \sum_{i=1}^{k_b-m-1} \begin{pmatrix} k_b - m - 1 \\ k_b - m - i \end{pmatrix} g_i \tilde{G}_b^{(k_b-m-i)}(-\lambda_b) + g_{k_b-m}. \tag{3.18}$$

Now, using Equations 3.13 and 3.17, $C_{b,m}$ can be expressed as

$$C_{b,m} = \frac{G_b(-\lambda_b) \times \tilde{G}_b^{(k_b-m)}(-\lambda_b)}{(k_b-m)!} \tag{3.19}$$

From Equations 3.12 and 3.19, we thus obtain

$$P_t\{s_0, s_1, \ldots, s_n\} = \sum_{b=1}^{a} \sum_{m=1}^{k_b} \frac{(\lambda_b t)^{m-1} e^{-\lambda_b t}}{(m-1)!} \times \frac{G_b(-\lambda_b) \times \tilde{G}_b^{(k_b-m)}(-\lambda_b)}{(k_b-m)!} \times \frac{Q_{s_0,s_1,\ldots,s_n}}{\lambda_b{}^{m-1}}.$$

Now assume that $Q_{b,s_0,s_1,\ldots,s_n} = G_b(-\lambda_b) \times Q_{s_0,s_1,\ldots,s_n}$, i.e.

$$Q_{b,s_0,s_1,\ldots,s_n} = \prod_{\substack{r=1 \\ r \neq b}}^{a} \frac{1}{(\lambda_r - \lambda_b)^{k_r}} \times Q_{s_0,s_1,\ldots,s_n}.$$

Accordingly,

$$P_t\{s_0, s_1, \ldots, s_n\} = \sum_{b=1}^{a} \sum_{m=1}^{k_b} \frac{(\lambda_b t)^{m-1} e^{-\lambda_b t}}{(m-1)!} \times \frac{\tilde{G}_b^{(k_b-m)}(-\lambda_b)}{(k_b-m)!} \times \frac{Q_{b,s_0,s_1,\ldots,s_n}}{\lambda_b{}^{m-1}}$$

Let

$$X_{s_0,s_1,\ldots,s_n}(b,m) = \frac{(\lambda_b t)^{m-1} e^{-\lambda_b t}}{(m-1)!} \times \frac{\tilde{G}_b^{(k_b-m)}(-\lambda_b)}{(k_b-m)!} \times \frac{Q_{b,s_0,s_1,\ldots,s_n}}{\lambda_b{}^{m-1}}.$$

This implies that

$$P_t\{s_0, s_1, \ldots, s_n\} = \sum_{b=1}^{a} \sum_{m=1}^{k_b} X_{s_0,s_1,\ldots,s_n}(b,m).$$

In the following Lemma, we present a result which allows us to write a recursive expression to compute $X_{s_0,s_1,\ldots,s_n}(b,m)$.

Lemma 3.2.2 *For $m = k_b$, $\tilde{G}_b^{(k_b-m)}(-\lambda_b) = 1$, and for $1 \le i \le k_b - 1$,*

$$\frac{\tilde{G}_b^{(k_b-m)}(-\lambda_b)}{(k_b-m)!} = \frac{1}{(k_b-m)} \left[\sum_{i=1}^{k_b-m-1} g_i^{'} \times \frac{\tilde{G}_b^{(k_b-m-i)}(-\lambda_b)}{(k_b-m-i)!} + g_{k_b-m}^{'} \right],$$

where

$$g_i^{'} = (-1)^i \sum_{r=1,\ r\neq i}^{a} \frac{k_r}{(\lambda_r - \lambda_b)^i}, \quad \textit{for } i \ge 1.$$

Proof: For $m = k_b$, it is obvious that $\tilde{G}_b^{(k_b-m)}(-\lambda_b) = 1$. For $1 \le m \le k_b - 1$, by considering Equation 3.18, we can write

$$\begin{aligned}
\frac{\tilde{G}_b^{(k_b-m)}(-\lambda_b)}{(k_b-m)!} &= \frac{1}{(k_b-m)!} \sum_{i=1}^{k_b-m-1} \begin{pmatrix} k_b-m-1 \\ k_b-m-i \end{pmatrix} g_i \tilde{G}_b^{(k_b-m-i)}(-\lambda_b) + g_{k_b-m} \\
&= \frac{1}{(k_b-m)!} \left[\begin{pmatrix} k_b-m-1 \\ k_b-m-1 \end{pmatrix} \times g_1 \times \tilde{G}_b^{(k_b-m-1)}(-\lambda_b) + \right. \\
&\quad \begin{pmatrix} k_b-m-1 \\ k_b-m-2 \end{pmatrix} \times g_2 \times \tilde{G}_b^{(k_b-m-2)}(-\lambda_b) + \cdots + \\
&\quad \left. \begin{pmatrix} k_b-m-1 \\ 1 \end{pmatrix} \times g_{k_b-m-1} \times \tilde{G}_b^{(1)}(-\lambda_b) + g_{k_b-m} \right] \\
&= \frac{1}{(k_b-m)} \left[\frac{g_1}{0!} \times \frac{\tilde{G}_b^{(k_b-m-1)}(-\lambda_b)}{(k_b-m-1)!} + \frac{g_2}{1!} \times \frac{\tilde{G}_b^{(k_b-m-2))}(-\lambda_b)}{(k_b-m-2)!} \right. \\
&\quad \left. + \cdots + \frac{g_{k_b-m-1}}{(k_b-m-2)!} \times \frac{\tilde{G}_b^{(1)}(-\lambda_b)}{1!} + \frac{g_{k_b-m}}{(k_b-m-1)!} \right]
\end{aligned}$$

$$= \frac{1}{(k_b - m)} \left[\sum_{i=1}^{k_b-m-1} g_i^{'} \times \frac{\tilde{G}_b^{(k_b-m-i)}(-\lambda_b)}{(k_b-m-i)!} + g_{k_b-m}^{'} \right] . \square$$

Using the Lemma 3.2.2, for $1 \leq m \leq k_b - 1$, we can express $X_{s_0,s_1,\ldots,s_n}(b,m)$ recursively as

$$X_{s_0,s_1,\ldots,s_n}(b,m) = \frac{1}{(k_b-m)} \left[\sum_{i=1}^{k_b-m} \frac{(m-1+i)}{t^i \times (m-1)!} \times X_{s_0,s_1,\ldots,s_n}(b,m+i) \times g_i^{'} \right], \quad (3.20)$$

where, for $m = k_b$,

$$X_{s_0,s_1,\ldots,s_n}(b,k_b) = \frac{t^{k_b-1}e^{-\lambda_b t}}{(k_b-1)!} \times \frac{Q_{b,s_0,s_1,\ldots,s_n}}{\lambda^{k_b-1}}.$$

3.3 Algorithm to Compute $P_t\{s_0, s_1, \ldots, s_n\}$

Recall that we can express the probability of occurence of a path, $P_t\{s_0, s_1, \ldots, s_n\}$, in terms of $X_{s_0,s_1,\ldots,s_n}(b,m)$ as

$$P_t\{s_0, s_1, \ldots, s_n\} = \sum_{b=1}^{a} \sum_{m=1}^{k_b} X_{s_0,s_1,\ldots,s_n}(b,m).$$

While the recursion developed in Equation 3.20 provides an efficient way of computing $X_{s_0,s_1,\ldots,s_n}(b,m)$, it is susceptible to numerical errors. Depending on the value of function $g_i^{'}$, the terms over which the summation is performed in Equation 3.20 may have both positive and negative signs. If these terms with different signs are close to each other, then their addition may cause a loss in precision. Our intent is to present a numerically stable algorithm by avoiding any losses in precision. To achieve this, we split $X_{s_0,s_1,\ldots,s_n}(b,m)$ into positive $X^p_{s_0,s_1,\ldots,s_n}(b,m) > 0$ and negative $X^n_{s_0,s_1,\ldots,s_n}(b,m) < 0$ parts such that

$$X_{s_0,s_1,\ldots,s_n}(b,m) = X^p_{s_0,s_1,\ldots,s_n}(b,m) + X^n_{s_0,s_1,\ldots,s_n}(b,m). \quad (3.21)$$

Similar to the recursion in Equation 3.20, for $1 \leq m \leq k_b - 1$, recursions for $X^p_{s_0,s_1,\ldots,s_n}(b,m)$ and $X^n_{s_0,s_1,\ldots,s_n}(b,m)$ can be expressed as

$$X^p_{s_0,s_1,\ldots,s_n}(b,m) = \frac{1}{(k_b-m)} \left[\sum_{i=1}^{k_b-m} \frac{(m-1+i)}{t^i \times (m-1)!} \times I(g_i^{'} > 0) \times g_i^{'} \times X^p_{s_0,s_1,\ldots,s_n}(b,m) + \sum_{i=1}^{k_b-m} \frac{(m-1+i)}{t^i \times (m-1)!} \times I(g_i^{'} < 0) \times g_i^{'} \times X^n_{s_0,s_1,\ldots,s_n}(b,m) \right] \tag{3.22}$$

and

$$X^n_{s_0,s_1,\ldots,s_n}(b,m) = \frac{1}{(k_b-m)} \left[\sum_{i=1}^{k_b-m} \frac{(m-1+i)}{t^i \times (m-1)!} \times I(g_i^{'} < 0) \times g_i^{'} \times X^p_{0,s_1,\ldots,s_n}(b,m) + \sum_{i=1}^{k_b-m} \frac{(m-1+i)}{t^i \times (m-1)!} \times I(g_i^{'} > 0) \times g_i^{'} \times X^n_{s_0,s_1,\ldots,s_n}(b,m) \right], \tag{3.23}$$

where $I(x)$ is an indicator function (returns 1 if x is true and else 0) and for $m = k_b$, $X^n_{s_0,s_1,\ldots,s_n}(b,m) = 0$ and

$$X^p_{s_0,s_1,\ldots,s_n}(b,k_b) = \frac{t^{k_b-1}e^{-\lambda t}}{(k_b-1)!} \times \frac{Q_{b,s_0,s_1,\ldots,s_n}}{\lambda^{k_b-1}}.$$

In terms of $X^p_{s_0,s_1,\ldots,s_n}(b,m)$ and $X^n_{s_0,s_1,\ldots,s_n}(b,m)$, using Equation 3.21, we can express path probability as

$$P_t\{s_0,s_1,\ldots,s_n\} = \sum_{b=1}^{a} \sum_{m=1}^{k_b} X^p_{s_0,s_1,\ldots,s_n}(b,m) + X^n_{s_0,s_1,\ldots,s_n}(b,m). \tag{3.24}$$

Since subtraction is the major cause for loss of precision in finite precision math, our goal is to avoid subtractions as much as possible. We achieve our goal by first adding the negative and positive terms separately and then by performing one necessary subtraction between them. Let $X^p_{s_0,s_1,\ldots,s_n}$ and $X^n_{s_0,s_1,\ldots,s_n}$ are the sum of these positive and negative numbers, respectively. Then by rearranging summations in Equation 3.24, we can write

$$X^p_{s_0,s_1,\ldots,s_n} = \sum_{b=1}^{a} \sum_{m=1}^{k_b} X^p_{s_0,s_1,\ldots,s_n}(b,m), \tag{3.25}$$

and

$$X^n_{s_0,s_1,\ldots,s_n} = \sum_{b=1}^{a}\sum_{m=1}^{k_b} X^n_{s_0,s_1,\ldots,s_n}(b,m), \tag{3.26}$$

which implies that

$$P_t\{s_0,s_1,\ldots,s_n\} = X^p_{s_0,s_1,\ldots,s_n} + X^n_{s_0,s_1,\ldots,s_n}. \tag{3.27}$$

If all additions are done in ascending order of absolute magnitudes, Equations 3.25 and 3.26 enable us to compute accurate (to the precision of the computation) values of $X^p_{s_0,s_1,\ldots,s_n}$ and $X^n_{s_0,s_1,\ldots,s_n}$. Until now we have successfully avoided subtractions. However, to compute the path probability (Equation 3.27) we have to perform the necessary subtraction between $X^p_{s_0,s_1,\ldots,s_n}$ and $X^n_{s_0,s_1,\ldots,s_n}$. Since this subtraction cannot be avoided, we need an alternative mechanism to ensure that the answer must be within the desired precision (in number of decimal digits). We do this by first checking the loss of precision in decimal digits caused by this subtraction. The sum of loss in precision and the desired accuracy of the answer gives us the required precision to perform subtraction. If the required precision is less than the precision provided by the floating point math (as per the IEEE standard), then the answer is acceptable. Otherwise, we add required precision and the provided precision to estimate an increase in precision which must be provided to obtain an accurate answer. Then, we roll back and invoke the multi-precision libraries to provide the estimated increased precision to recompute all intermediate terms and the final subtraction. Since we get more accurate operands for the final subtraction, we may see an increase in the loss in precision. So we again check that if at this stage the required precision is greater than the provided precision. If it is, we again roll back and recompute all intermediate terms and the final subtraction with more enhanced precision. This checking continues until the required precision becomes less than the provided precision that is when we get an answer with the desired accuracy.

More precisely, our algorithm to compute probability of occurence of a path $P\{s_0, s_1, \ldots, s_n\}$ is as follows. (Implicitly, all additions in the algorithm are performed in ascending order of absolute magnitudes.)

Algorithm 3.3.1 *(Compute the probability of occurence of a path $P\{s_0, s_1, \ldots, s_n\}$ using Equations 3.22, 3.23, 3.25, 3.26, and 3.27.)*

IEEE_precision = precision provided as per IEEE standard.

Compute $X^p_{s_0,s_1,\ldots,s_n}(b)$ using Equations 3.22 and 3.25.

Compute $X^n_{s_0,s_1,\ldots,s_n}(b)$ using Equation 3.23 and 3.26.

result = $X^p_{s_0,s_1,\ldots,s_n}(b)\ +\ X^n_{s_0,s_1,\ldots,s_n}(b)$.

By looking at the exponents of $X^p_{s_0,s_1,\ldots,s_n}(b)$, $X^n_{s_0,s_1,\ldots,s_n}(b)$, and result*, find the loss_in_precision by performing the necessary subtraction.*

required_precision = desired_precision + loss_in_precision.

provided_precision = IEEE_precision.

While(required_precision > provided_precision)

provided_precision = required_precision + IEEE_precision.

Recompute $X^p_{s_0,s_1,\ldots,s_n}(b,m)$ *and* $X^n_{s_0,s_1,\ldots,s_n}(b,m)$ *with the estimated provided provision by invoking multi-precision routines.*

result $= X^p_{s_0,s_1,\ldots,s_n}(b) + X^n_{s_0,s_1,\ldots,s_n}(b)$.

By looking at the exponents of $X^p_{s_0,s_1,\ldots,s_n}(b)$, $X^n_{s_0,s_1,\ldots,s_n}(b)$, *and result, find the loss_in_precision by performing the necessary subtraction.*

required_precision = desired_precision + loss_in_precision.

End(While).

$P\{s_0, s_1, \ldots, s_n\}$ = *result.*

The algorithm, together with the path-based approach described in Section 3.1, provides the basis for a stable computation of state-occupancy probabilities at a given instant in time. The overall algorithm to compute instant-of-time measures is based on generating paths in the continuous-time Markov process, computing their probabilities, and summing up the occupancy probabilities of paths whose last states have identical rate rewards. These

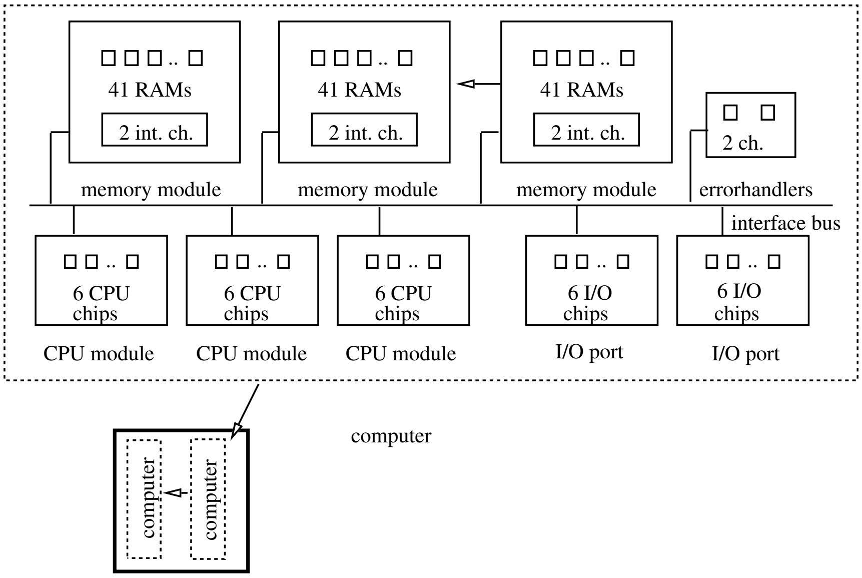

Figure 3.1: Fault-tolerant parallel computing system

probabilities corresponding to the assigned rate rewards are then used in Equation 3.1 and 3.2 to compute expected value and distribution of an instant-of-time measure, respectively.

To test the algorithm on a non-trivial example, we have implemented it using a C++ multi-precision library [43].

3.4 Example

The applicability of our approach is illustrated by considering the reliability analysis of a highly-redundant fault-tolerant multiprocessor system from Lee *et al.* [53]. The system, at the highest level, consists of two non-repairable computers (Lee et al. considered a variable number of computers.) (Note that our algorithm applies to repairable, as well as non-repairable systems.) As shown in Figure 3.1, each computer is composed of three memory modules, of which one is a spare; three CPU units, of which one is spare; two I/O ports, of which one is spare; and two non-redundant error handling chips. A computer is classified as operational if at least two memory modules, two CPU units, one I/O port, and the two error-handling chips are working.

Internally, each memory module consists of 41 RAMs, out of which two are spare chips and two are interface chips. Each CPU unit and each I/O port consists of six non-redundant chips. A memory module is classified as operational if at least 39 of its 41 RAM chips and its two interface chips are working. Each component failure has coverage probabilities associated with it. The coverage probabilities are state dependent, and there are multiple levels of coverage (module, computer, multiprocessor). We used the same coverage probabilities and the failure rate for the chips as used in the original model by Lee et al. [53].

The model is represented as a stochastic activity network as was done in [54]. The state-level representation of the model consists of 10114 states. The model is solved on a HP-715/64 workstation using the path-based algorithm developed in this chapter to compute the reliability measure. The algorithm is implemented such that the discarding criteria is checked for every path whose length is greater than 3 states. (All paths which can be generated by 3 or more transitions of the Markov process).

The results presented in this section are significant in two ways. First, they show that the path-based algorithm developed in this chapter can be used to compute instant-of-time measures and, second, they show that due to the nature of the computation of path probabilities, for even a reasonably small number of paths (tens of thousands), the algorithm takes considerable amount of time. Specifically, graphs in Figures 3.2 and 3.3 show the validity of the algorithm. The graph in Figure 3.2 shows the reliability values computed at instants of 1, 2, 3, 4, and 5 years. Whereas, the graph in Figure 3.3 illustrates the upper bounds on the error induced in the reliability values by considering a subset of the total number of paths in the continuous time Markov process. (Recall that the algorithm truncates a path if the sum of probabilities of all the paths that can be generated from it is less than weight $w = 10^{-10}$.)

Graphs in Figure 3.4 show that the execution time for even a reasonably low number of considered paths is considerably high. For example, the execution time is 7.8 hrs for $51,000$ considered paths. The reason for high execution time is the extensive use of multi-precision library routines. In this example, due to the nature of the values of outgoing rates from the states of the continuous-time Markov process, an accuracy of around 24 decimal digits is required for the calculation of probabilities of almost all the paths. This implies that multi-precision libraries are extensively used since the IEEE floating point standard can only provide an accuracy of 16 decimal digits.

We would like to conclude this chapter by noting that due to the high execution this method does not appear to be useful. However, this method has a potential to become competitive when the extended IEEE floating point standard (which gives an accuracy of 32 decimal digits) becomes available. Compilers on the new 64-bit architectures consider long double (which is defined in the extended IEEE floating point standard) as a valid data type but they do not have hardware support for it. In the near future, the 64-bit architectures will support the extended IEEE floating point standard at the hardware level. Then the execution time of the path-based approach will reduce drastically.

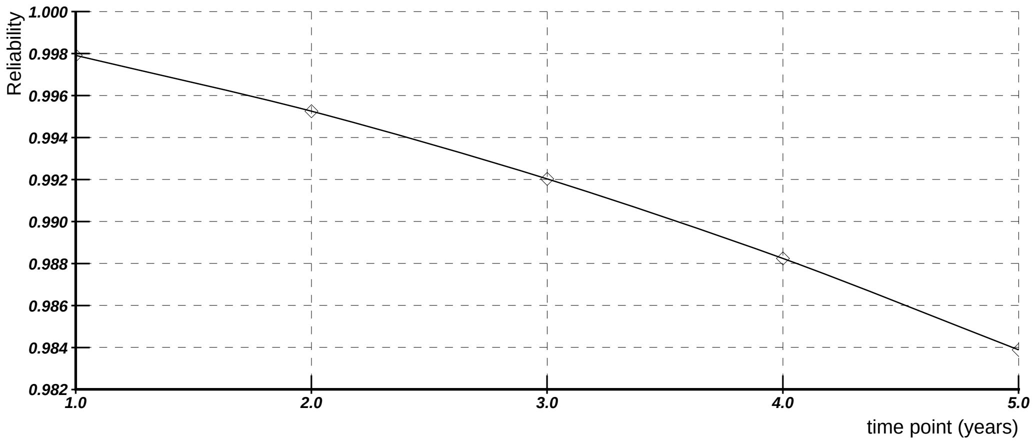

Figure 3.2: Reliability

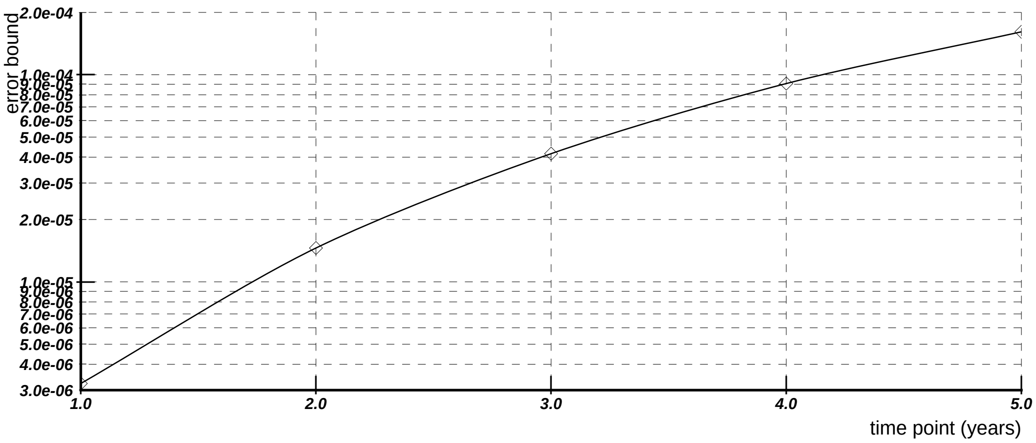

Figure 3.3: Error bound

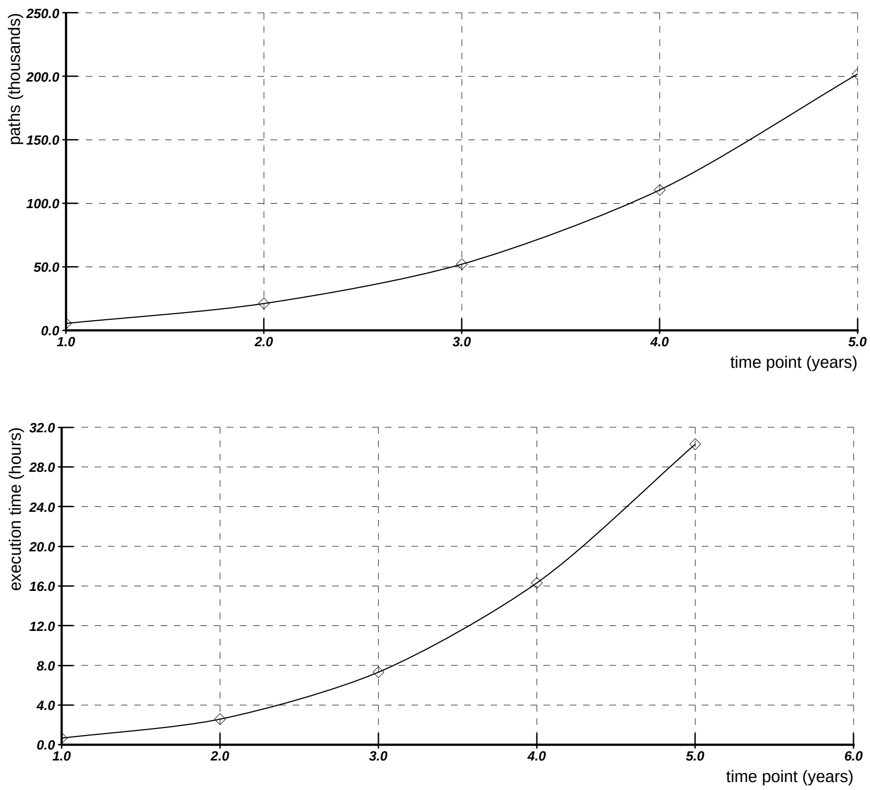

Figure 3.4: Effect of increasing paths on execution time

CHAPTER 4

Distribution of Reward Accumulated During a Finite Interval

The last chapter presented a path-based approach to analyze system characteristics such as reliability and instantaneous availability. In addition to these instant-of-time system characteristics, it is very crucial to know about the overall performance of the system during its mission time. This chapter presents solution of time-averaged interval-of-time variables which provide such insights in this regard. Specifically, we present a path-based approach to compute the probability distribution function (PDF) of time-averaged reward accumulated over a finite interval for general (possibly cyclic) reward models. In doing so, we trade storage complexity for time complexity. We show that the trade is advantageous if the probability mass is concentrated on a reasonable number of paths (up to tens of millions). Moreover, to use the path-based approach effectively, we present a numerically stable and computationally efficient algorithm to compute the conditional distribution of accumulated reward given a path. This formulation is new and not based on the often-used Weisberg result. Finally, we illustrate the usefulness of the path-based approach by computing the distribution of time-averaged accumulated reward over a finite interval of a highly redundant fault-tolerant multiprocessor system. The new approach presented in this chapter is significant in that it can be applied to much larger systems than previously possible, and it is numerically stable.

The remainder of the chapter is organized as follows. Section 4.1 reviews material needed in the computation of the PDF of accumulated reward, and in its context, highlights why the path-based approach is needed. Section 4.2 formalizes our approach and derives a bound on the error induced by only considering finite many paths. Section 4.3 then develops a formulation to compute the conditional distribution given a path, and Section 4.4 presents a numerical stable algorithm by using this formulation. Section 4.5 illustrates the effectiveness of a path-based approach by solving a large state-space example.

4.1 Background

Uniformization (also known as Jenson's method and randomization) is a well-known method for computing the state-occupancy probabilities of a Markov process at specific time t. Its formulation involves construction of a discrete-time Markov process and Poisson process from an original continuous-time process. In particular, consider a continuous-time time-homogeneous Markov process $\{X_t : \ t \geq 0\}$ with generator matrix A and initial state distribution vector π_0. Let λ be greater than or equal to the maximum departure rate from any state in the process, and $\{Z_n : \ n \in N\}$ be a discrete-time time-homogeneous Markov chain defined on the same state space as $\{X_t : \ t \geq 0\}$ with single step transition matrix $P = A/\lambda + I$. Furthermore, let $\{N_t : \ t \geq 0\}$ be a Poisson process with rate λ. Then π_t, the row vector of state occupancy probabilities at time t, can be expressed as

$$\pi_t \ = \ \sum_{n=0}^{\infty} \frac{e^{-\lambda t}(\lambda t)^n}{n!} \pi_0 P^n.$$

de Souza e Silva and Gail [36, 44] formulated the probability distribution function (PDF) of the accumulated reward averaged over a finite interval of time using uniformization and several additional properties about a Markov chain subordinated to a Poisson process. Specifically, consider that the Poisson process has arrivals at instances $T_1 < T_2 < \ldots < T_n$ in an interval $(0, t)$. These are the instances when the subordinated Markov chain makes transitions. Furthermore, consider n independent random variables $U_1, U_2, \ldots U_n$ uniformly distributed on the interval $(0, 1)$. Let $U_{(1)}$, $U_{(2)}$, …, $U_{(n)}$ be their order statistics. It is well known (e.g., [45]) that the joint distribution of the transition times of subordinated Markov chains, given n transitions of the Poisson process in an interval $(0, t)$, is identical to the joint distribution of the order statistics of n uniformly distributed random variables over the interval $(0, t)$. Moreover, it can easily be shown that tU_i, the product of the interval t and the random variable U_i, is uniformly distributed over an interval $(0, t)$. Therefore

$$P\{T_1 \leq t_1, T_2 \leq t_2, \ldots, T_n \leq t_n\} \;=\; P\{tU_{(1)} \leq t_1, tU_{(2)} \leq t_2, \ldots, tU_{(n)} \leq t_n\}, \tag{4.1}$$

for $t_1 < t_2 < \ldots < t_n$.

Given n transitions of the Poisson process, each path of the subordinated Markov chain consists of $n+1$ sojourn times. Using the result from Equation (4.1), these $n+1$ sojourn times Y_i can be represented as $Y_1 = tU_{(1)}$, $Y_2 = t(U_{(2)} - U_{(1)})$, …, $Y_{n+1} = t(1 - U_{(n)})$. It is also known [46] that the random variables Y_i, $i = 1, 2, \ldots, n+1$, are exchangeable. In particular, exchangeability says that

$$P\{Y_1 \leq t_1, Y_2 \leq t_2, \ldots, Y_{n+1} \leq t\} \;=\; P\{Y_{j_1} \leq t_1, Y_{j_2} \leq t_2, \ldots, Y_{j_{n+1}} \leq t\},$$

for all permutations of j_i's from $1, 2, \ldots, n+1$. Therefore, by using order statistics, the sequence of sojourn times in a state trajectory can be altered. In particular, suppose all paths with n transitions of the uniformized process are divided into sets such that each path in a set has an equal number of visits to states with identical rate rewards. Then exchangeability enables us to describe all paths within each set by a vector $\mathbf{k} = (k_1, k_2, \ldots, k_{K+1})$, where $K+1$ is the number of distinct rate rewards and k_i, $i = 1, 2, \ldots, K+1$, is equal to the number of visits to states with rate reward i. (Note $|\mathbf{k}| = n+1$.)

Using these ideas, de Souza e Silva and Gail formulated the PDF of reward accumulated over a finite interval by further conditioning on the vectors $\mathbf{k}$ possible for each n. Accordingly, $P\{AR(t) \leq r\}$, the distribution of the time-averaged reward accumulated over a finite interval, was expressed as

$$P\{AR(t) \leq r\} = \sum_{n=0}^{\infty} \frac{e^{-\lambda t}(\lambda t)^n}{n!} \sum_{\forall \mathbf{k}} P\{\mathbf{k} \mid n\} P\{AR(t) \leq r \mid n, \mathbf{k}\}, \tag{4.2}$$

where $P\{\mathbf{k} \mid n\}$ is the probability of vector $\mathbf{k}$ given n transitions.

The exact solution for $P\{AR(t) \leq r\}$ is not possible in this formulation, since the first summation in (4.2) is over an infinite set. However, as shown in [36, 44], a solution with error-bound can be computed by truncating the first summation to some finite number N. In particular, to a certain error-bound, the probability distribution of time-averaged reward accumulated over a finite interval can be formulated as

$$P\{AR(T) \leq r\} = \sum_{n=0}^{N} \frac{e^{-\lambda t}(\lambda t)^n}{n!} \sum_{\forall \mathbf{k}} P\{\mathbf{k} \mid n\} P\{AR(T) \leq r \mid n, \mathbf{k}\}. \tag{4.3}$$

For Markov processes with large state spaces, second summation is challenging to compute due, primarily, to the computation of $P\{\mathbf{k} \mid n\}$. This computation requires knowledge of all $\mathbf{k}$ vectors that are possible given n transitions of the uniformized process and the probability of occurrence of paths within these sets of vectors. Since there can be $\begin{pmatrix} K+n \\ n \end{pmatrix}$ $\mathbf{k}$ vectors for n transitions and the number of paths which can generate a vector $\mathbf{k}$ can be equal to the size of the state space, the task of computing PDF for large state spaces becomes immensely difficult. Moreover, this approach also requires the complete generation of the state space prior to starting the PDF computation. Generation of large state spaces in itself is an intensively memory complex problem.

In [38], de Souza e Silva and Gail improved on their earlier algorithm by finding a recursion which, for a given number of transitions, depends on sets of vectors $\mathbf{k}$. Instead of computing path probabilities for each vector $\mathbf{k}$, they directly computed the conditional distribution given n for sets of $\mathbf{k}$ vectors. Their new algorithm significantly improved the memory complexity since individual $\mathbf{k}$ vectors no longer need to be computed. However, due to the recursion on sets of $\mathbf{k}$ vectors, their algorithm needs a separate computation for every point on the probability distribution curve. When both rate and impulse rewards are used, their algorithm has $O(dMN^2\kappa(N,r))$ storage requirement [47], where d is the average number of non-zero entries in the transition matrix, M is the state-space size, N is the truncation point of the infinite series, and $\kappa(N,r)$ is the number of distinct values obtained by adding any combination of N impulse rewards that are less than r. Furthermore, their algorithm operates on the state transition matrix of the subordinated Markov process, which also must be stored. While the new algorithm takes significantly less memory than the original algorithm, it still requires prohibitively large amounts of memory for realistically sized models, which may have hundreds of thousands of states. The next section presents a formulation that avoids this problem, albeit at the cost of additional time, and avoids numerical difficulties in the formulation presented in [36].

4.2 Path-Based Algorithm

In this new formulation, we compute the probability distribution function of the time-averaged reward accumulated over a finite interval of time by conditioning on possible paths, rather than **k** vectors, that can occur. These paths are generated in a depth-first search manner, from a higher level description (such as a SAN [2]), thus avoiding the memory complexity problems associated with previous approaches. However, this gain is not free, since the approach results in an increased time complexity. In the worst case, the number of paths of the uniformized process that must be generated grows exponentially with an increase in the truncation point. However, if the uniformized process has a sparse state-transition probability matrix or is such that the probability mass is concentrated on a reasonable number of finitely many paths, then the depth-first search approach becomes a reasonable choice.

In this case, we can compute the solution with a reasonable error bound by only considering the set of probable paths. Moreover, if we know the highest rate of the underlying Markov process (as would be the case if it were generated from a higher-level formalism), then the depth-first search approach does not even require generation of the state space prior to the solution. In this case, the paths of the uniformized process can be explored without generating the complete state space, and the solution for the time-averaged accumulated reward can be obtained on-the-fly.

The depth-first search approach for computing the PDF of the time-averaged reward accumulated over a finite interval is based on exploring paths of the uniformized process and computing conditional distribution given the path. To understand this approach, we first look at a path in the uniformized process. A path in the uniformized process corresponds to a sequence of states through the state space. Let $< s_0, s_1, \ldots, s_n >$ be a sequence in the uniformized process that occurs in time $(0, t)$. The probability of its corresponding path in time $(0, t)$ can be obtained by computing $P\{s_0, s_1, \ldots, s_n \mid n\} = p(s_0, s_1) \times p(s_1, s_2) \times \ldots p(s_{n-1}, s_{n-1})$, and $P\{n\}$,

$$P\{path\} \;=\; P\{s_0, s_1, \ldots, s_n \;\wedge\; n\} \;=\; P\{n\}P\{s_0, s_1, \ldots, s_n \mid n\}$$

where (s_i, s_j) is the transition probability from state s_i to s_j in the discrete time embedded process.

For the path-based approach, the distribution of the time-averaged reward accumulated over a finite interval can be expressed as

$$P\{AR(t) \leq r\} = \sum_{path \in P} P\{path\}P\{AR(t) \leq r | path\}, \tag{4.4}$$

where P is the (possibly infinite) set of possible paths in the process.

As with the formulation in (4.2), an exact solution of (4.4) is not possible because there are an infinite number of paths in a uniformized process. Since our intent is to only consider paths that are significant relative to the solution, we limit the number of paths considered by discarding those whose path probabilities are smaller than a specified value, defined as w. Let P_w denote the set of paths that have path probabilities greater than or equal to w. Accordingly, PDF can be expressed as

$$P\{AR(t) \leq r\} = \sum_{path \in P_w} P\{path\}P\{AR(t) \leq r | path\}. \tag{4.5}$$

As with the previous approach, considering a finite number of paths results in an error in the solution which can be bounded. In particular, let $E(w)$ be the error induced by discarding paths for which $P\{path\} < w$ and $0 \leq w \leq 1$. In order to bound $E(w)$, we first compute $E(path)$, the error produced by discarding a particular path. Let this path consist of the sequence $< s_0, s_1, \ldots, s_n >$. Note that by discarding a path, we are also discarding all longer paths which have states $s_0, s_1, \ldots, s_n$ as their first n states. We first look at the error induced by discarding the path of length n. In particular, the error will be

$$E(path) \;=\; \frac{e^{-\lambda t}(\lambda t)^n}{n!} P\{s_0, s_1, \ldots, s_n \mid n\} P\{AR(t) \leq r | path\}.$$

Since the conditional distribution $0 \leq P\{AR(t) \leq y | path\} \leq 1 \; \forall y$, we can bound the error by assuming it is equal to 1. This implies that

$$E(path) \;\leq\; \frac{e^{-\lambda t}(\lambda t)^n}{n!} P\{s_0, s_1, \ldots, s_n \mid n\}.$$

Now, since the entries in a row of a transition matrix sum to 1, the sum of the probabilities of all paths of length $n+1$ discarded due to the discarding of the path with state sequence $< s_0, s_1, \ldots, s_n >$ can be bounded by

$$\sum_{\forall s_{n+1}} P\{s_0, s_1, \ldots, s_{n+1} \mid n+1\} \times \frac{e^{-\lambda t}(\lambda t)^{n+1}}{(n+1)!} \;=\; P\{s_0, s_1, \ldots, s_n \mid n\} \frac{e^{-\lambda t}(\lambda t)^{n+1}}{(n+1)!}.$$

Using the above argument repeatedly, the total error (denoted E^*) induced by discarding all paths of length n or longer with starting states $s_0, s_1, \ldots s_n$ can be bounded by

$$\begin{aligned} E^*(path) &\leq P\{s_0, s_1, \ldots, s_n \mid n\} \times \sum_{i=n}^{\infty} \frac{e^{-\lambda t}(\lambda t)^i}{i!} \\ &= P\{s_0, s_1, \ldots, s_n \mid n\} \times \left(1 - \sum_{i=0}^{n-1} \frac{e^{-\lambda t}(\lambda t)^i}{i!}\right). \end{aligned}$$

Hence a bound on the total error induced by discarding paths can be computed as

$$E(w) \;\leq\; \sum_{path \in P^D(w)} E^*(w, path),$$

where $P^D(w)$ is the set of paths discarded, during exploration, since they do not meet the criteria described.

4.3 Calculation of Conditional Distribution

Given the approach of the previous section, we need a way to compute the distribution of time-averaged reward accumulated over an interval of time conditioned on a particular path. Finding an efficient and numerically stable way to compute this distribution is the key to the development of a methodology that can be applied to realistically sized systems. Since our process is uniformized with paths that are exchangeable (see Section II), the conditional accumulated reward will be the same for all paths that have the same number of visits to states with particular rate rewards. In other words, for all paths with $\mathbf{k}$ vector $\mathbf{k} = k_1, k_2, \ldots, k_{K+1}$ such that $|\mathbf{k}| = n + 1$,

$$P\{AR(t) \leq r \mid path\} \;=\; P\{AR(t) \leq r \mid n, \mathbf{k}\}.$$

This suggests that, in principle, we could use the result from Weisberg [48], which gives the probability distribution function of a linear combination of selected order statistics of i.i.d. uniform random variables. Unfortunately, while this is correct, it is numerically unstable for practical problems, since for large n, it requires the subtraction of extremely large numbers (generated from factorials of large numbers) with nearly identical magnitude at multiple points in the algorithm. Multiple precision arithmetic could potentially be used to avoid this problem, but it is extremely difficult to know what precision to use, because of the multiple subtractions.

To avoid these problems, we have developed a new algorithm that does not make use of the Weisberg result. It is based on an alternative formulation given in [49] and makes use of three new lemmas (presented in the following) that reduce both the amount of computation and the magnitude of numbers that must be stored as intermediate results. Furthermore, we are able to formulate the expression in a form that requires very few subtractions of numbers with large but nearly identical magnitude and, hence, are able to determine exactly the number of digits required to achieve a desired precision in the result.

In particular, the new formulation is based on expressing the problem as the linear combination of order statistics, rewriting these as a linear combination of Drichlet random variables, and then computing the distribution of the combination. (The Drichlet distribution is widely used in statistical mathematics, see [50] for example.) Specifically, consider, as was done previously, that $K+1$ different rate rewards are assigned to the states of the uniformized process. Furthermore, suppose that rate rewards are ordered such that

$$r_1 > r_2 > \ldots > r_{K+1} \geq 0.$$

Then define l_i to be the sum of the lengths of sojourn times with rate reward r_i. After doing this, the conditional averaged accumulated reward, $AR(t)$, given n and $\mathbf{k}$, can be expressed as

$$AR(t \mid n, \mathbf{k}) \;=\; \frac{1}{t}\sum_{i=1}^{K+1} r_i \times l_i.$$

Now recall that, using the result from (4.1), the $n+1$ sojourn times Y_i can be represented as $Y_1 = tU_{(1)}$, $Y_2 = t(U_{(2)} - U_{(1)})$, …, $Y_{n+1} = t(1 - U_{(n)})$. Resultingly, $P\{Y_1 + Y_2 \leq r\} = P\{tU_2 \leq r\}$. According to the exchangeability property, given n transitions, $P\{Y_{j_1} + Y_{j_2} + \ldots + Y_{j_m} \leq r\} = P\{tU_m \leq r\}$ for all permutations $j_1, j_2, \ldots, j_m$, $m \leq n+1$, of

$1, 2, \ldots, n+1$. Therefore, we can rearrange the sojourn times in a state trajectory such that first k_1 intervals are of rate r_1, the next k_2 intervals are of rate r_2, and so on. Then, sum of the sojourn times for rates rewards r_i, $i = 1, 2, \ldots, K+1$, can be expressed as

$$
\begin{aligned}
&l_1 = Y_1 + Y_2 + \ldots + Y_{k_1},\\
&l_2 = Y_{k_1+1} + Y_{k_1+2} + \ldots + Y_{k_1+k_2},\\
&\vdots\\
&l_{K+1} = Y_{k_1+\cdots+k_K+1} + Y_{k_1+\cdots+k_K+2} + \ldots + Y_{k_1+\cdots+k_{K+1}}.
\end{aligned}
$$

Given this, the conditional distribution function can be formulated as

$$
P\{AR(t) \le r \mid n, \mathbf{k}\} \;=\; P\left\{ \frac{1}{t} \times \begin{bmatrix} r_1(Y_1 + \ldots + Y_{k_1}) + \\ r_2(Y_{k_1+1} + \ldots + Y_{k_1+k_2}) + \\ \vdots \\ r_{K+1}(Y_{k_1+\cdots+k_K+1} + \ldots + Y_{k_1+\cdots+k_{K+1}}) \end{bmatrix} \le r \right\}.
$$

Furthermore, by scaling the rate rewards such that $b_i = r_i - r_{K+1}$, for $i = 1, 2, \ldots, K+1$, the conditional distribution can be expressed as

$$
P\{AR(t) \le r \mid n, \mathbf{k}\} \;=\; P\left\{ \frac{1}{t} \times \begin{bmatrix} b_1(Y_1 + \ldots + Y_{k_1}) + \\ b_2(Y_{k_1+1} + \ldots + Y_{k_1+k_2}) + \\ \vdots \\ b_K(Y_{k_1+\cdots+k_{K-1}+1} + \ldots + Y_{k_1+\cdots+k_K}) \end{bmatrix} \le r - r_{K+1} \right\}.
$$

Now define the random variables V_i such that

$$
\begin{aligned}
V_1 &= \tfrac{Y_1 + \ldots + Y_{k_1}}{t},\\
V_2 &= \tfrac{Y_{k_1+1} + \ldots + Y_{k_1+k_2}}{t},\\
&\vdots\\
V_K &= \tfrac{Y_{k_1+\cdots+k_{K-1}+1} + \ldots + Y_{k_1+\cdots+k_K}}{t}.
\end{aligned}
$$

As given in [49], these random variables $(V_1, V_2, \ldots, V_K)$ have a K-variate Drichlet probability density function (pdf)

$$f(v_1, \ldots, v_K) = \frac{\Gamma(n+1)}{\prod_{i=1}^{K+1} \Gamma(k_i)} \prod_{i=1}^{K} (v_i)^{(k_i-1)} \left\{1 - \sum_{i=1}^{K} v_i\right\}^{(k_{K+1}-1)},$$

at any point in the simplex:

$$\left\{(v_1, \ldots, v_K) : v_i \geq 0, i = 1, \ldots, K, \sum_{i=1}^{K} v_i \leq 1\right\},$$

in the K-dimensional real space and zero outside.

Using these random variables, we can then express the conditional time-averaged accumulated reward as

$$P\{AR(t) \leq r \mid n, \mathbf{k}\} = P\left\{\sum_{i=1}^{K} b_i V_i \leq b\right\}, \text{ where } b = r - r_{K+1}. \tag{4.6}$$

Given (4.6), the problem of computing conditional time-averaged accumulated reward is reduced to the computation of the distribution of a linear combination of random variables with K-variate Drichlet distribution. To compute this, we use a result given in [49] for the density function of this distribution. In particular, the density function of $AR(t)$, given $\mathbf{k}$ and n is

$$f_{AR(t)}(b \mid n, \mathbf{k}) = \prod_{a=1}^{K} b_a^{-k_a} \sum_{l=1}^{K} \sum_{m=1}^{k_l} C_{l,m} \frac{\chi(b/b_l)\chi(1-(b/b_l)) b^{m-1} (1-(b/b_l))^{n-m}}{B(m, n-m+1)}, \tag{4.7}$$

where

$$\chi(z) = \begin{cases} 1, & \text{if } z > 0, \\ 0, & \text{else,} \end{cases}$$

$$B(i,j) = \tfrac{\Gamma(i)\Gamma(j)}{\Gamma(i+j)},$$

and $C_{l,m}$ are constant coefficients for the partial fraction of

$$G(s) = \frac{1}{(s+\beta_1)^{k_1}(s+\beta_2)^{k_2} \ldots (s+\beta_k)^{k_K}},$$

where $\beta_i = (1/b_i)$, $i = 1, 2, \ldots, K$, are zeros of $1/G(s)$ and k_i, $i = 1, 2, \ldots, K$, denote their order.

The PDF of $AR(t)$, given n and $\mathbf{k}$, is thus (by integration)

$$F_{AR(t)}(b \mid n, \mathbf{k}) = \prod_{a=1}^{K} b_a^{-k_a} \sum_{l=1}^{K} \sum_{m=1}^{k_l} \frac{C_{l,m}}{B(m, n-m+1)} \int_0^b \chi(s/b_l)\chi(1-(s/b_l)) s^{m-1}(1-(s/b_l))^{n-m} ds. \tag{4.8}$$

To evaluate the integral in (4.8), we must know the regions in which the unit step functions have a non-zero value. Note that

$$\chi(s/b_l) = \begin{cases} 1 & if\ s > 0, \\ 0 & else. \end{cases}, \text{ and } \quad \chi(1-(s/b_l)) = \begin{cases} 1 & if\ s < b_l, \\ 0 & else. \end{cases}.$$

Thus

$$F_{AR(t)}(b \mid n, \mathbf{k}) = \prod_{a=1}^{K} b_a^{-k_a} \sum_{l=1}^{K} \sum_{m=1}^{k_l} \frac{C_{l,m}}{B(m, n-m+1)} \begin{cases} \int_0^{b_l} s^{m-1}(1-(s/b_l))^{n-m} ds & if\ b > b_l \\ \int_0^{b} s^{m-1}(1-(s/b_l))^{n-m} ds & else. \end{cases}$$

After solving the integrals

$$F_{AR(t)}(b \mid n, \mathbf{k}) = \prod_{a=1}^{K} b_a^{-k_a} \sum_{l=1}^{K} \sum_{m=1}^{k_l} C_{l,m} \begin{cases} b_l^m & if\ b \geq b_l \\ \frac{b^m}{m} \frac{H[m, m-n, m+1, (b/b_l)]}{B(m, n-m+1)} & if\ 0 \leq b < b_l \end{cases}, \tag{4.9}$$

where $H[m, m-n, m+1, (b/b_l)]$ is a hyper-geometric function defined by the following series,

$$1 + \sum_{u=1}^{\infty} \frac{m(m+1)\ldots(m+u-1)\ (m-n)(m-n+1)\ldots(m-n+u-1)}{u!\ (m+1)(m+2)\ldots(m+u)} \left(\frac{b}{b_l}\right)^n. \tag{4.10}$$

Similarly, the complementary probability distribution function of $AR(t)$, given $\mathbf{k}$ and n, denoted ($\overline{F}_{AR(t)}(b \mid n, \mathbf{k})$), can be computed as

$$\overline{F}_{AR(t)}(b \mid n, \mathbf{k}) = \begin{cases} 0 & if\ b \geq b_l \\ \prod_{a=1}^{K} b_a^{-k_a} \sum_{l=1}^{K} \sum_{m=1}^{k_l} C_{l,m} \left(b_l^m - \frac{b^m}{m} \frac{H[m, m-n, m+1, (b/b_l)]}{B(m, n-m+1)} \right) & if\ 0 \leq b < b_l \end{cases}. \tag{4.11}$$

In the Equation (4.11), we assume that $0 \leq k_l \leq n$ such that $\sum_{l=1}^{K} k_l = n+1$. When $k_l = n+1$, the Markov chain only visits states with rate reward r_l. The conditional probability distribution is then 1, if the rate reward b_l is greater than b, or 0, if it is less. More generally, suppose x out of K rate rewards are greater than b, i.e., $b_1 > b_2 > \cdots > b_x > b$, then

$$\overline{F}_{AR(t)}(b \mid n, \mathbf{k}) = \prod_{a=1}^{K} b_a^{-k_a} \sum_{l=1}^{x} \sum_{m=1}^{k_l} C_{l,m} \times b_l^m \times \left(1 - \left(\frac{b}{b_l}\right)^m \frac{H[m, m-n, m+1, (b/b_l)]}{m\ B(m, n-m+1)}\right) \tag{4.12}$$

By looking at (4.12), one can easily realize that its computation by straightforward means, like the Weisberg result, is susceptible to numerical errors. The main reason for these errors is the computation of factorials in computing $H[m, m-n, m+1, (b/b_l)]$, $B(m, n-m+1)$, and $C_{l,m}$. These expressions result in extremely large numbers which when subtracted using normal floating point arithmetic may introduce significant loss of precision. However, with appropriate manipulation, and selective use of extended precision math, these problems can be avoided. In the remainder of this section, we derive a computationally efficient and numerically stable means to compute $\overline{F}_{AR(t)}(b \mid n, \mathbf{k})$.

For simplicity, in the remaining discussion, let

$$F_l(n,m) = \left(1 - \left(\frac{b}{b_l}\right)^m \frac{H[m, m-n, m+1, (b/b_l)]}{m\ B(m, n-m+1)}\right). \tag{4.13}$$

Accordingly,

$$\overline{F}_{AR(t)}(b \mid n, \mathbf{k}) = \prod_{a=1}^{K} b_a^{-k_a} \sum_{l=1}^{x} \sum_{m=1}^{k_l} C_{l,m} \times b_l^m \times F_l(n,m),$$

which after algebraic manipulation can be written as

$$\overline{F}_{AR(t)}(b \mid n, \mathbf{k}) = \sum_{l=1}^{x} \prod_{a=1, a\neq l}^{K} b_a^{-k_a} \sum_{m=1}^{k_l} \frac{1}{b_l^{(k_l-m)}} \times C_{l,m} \times F_l(n,m).$$

4.3.1 Computation of $C_{l,m}$

To compute $C_{l,m}$, the constant coefficients of the partial fraction expansion of $G(s)$, we need to compute higher derivatives of the function $G_l(s)$ (the ith derivative of G_l is denoted $G_l^{(i)}$), where

$$G_l(s) \;\equiv\; (s+\beta_l)^{k_l} G(s) \;=\; \prod_{a=1, a\neq l}^{K} \frac{1}{(s+\beta_a)^{k_a}},$$

since

$$C_{l,m} \;=\; \lim_{s\to(-\beta_l)} \frac{1}{(k_l-m)!}\, G_l^{(k_l-m)}(s) \;\equiv\; \frac{1}{(k_l-m)!}\, G_l^{(k_l-m)}(-\beta_l). \tag{4.14}$$

For computing higher derivatives, we take the same approach as detailed in Section 3.2.1. Specifically, we use Lemma 3.2.1 by defining $f(s) = \exp(s)$, $h(s) = f(g(s))$, and

$$g(s) \;=\; -\sum_{a=1, a\neq l}^{K} k_a \ln(s+\beta_a).$$

Note that $h(s) = \exp(g(s)) = G_l(s)$. Furthermore, observe that

$$\frac{d^{(p)}}{dg} G_l(g(s)) \;=\; \exp(g(s)) \;=\; G_l(-\beta_l) \;=\; \prod_{a=1, a\neq l}^{K} (\beta_a - \beta_l)^{-k_a}, \tag{4.15}$$

for $p \geq 1$. Using Lemma 1, $G^{(k_l-m)}(-\beta_l) = h_{k_l-m}$ can then be expressed as

$$G_l^{(k_l-m)}(-\beta_l) \;=\; \sum_{i=1}^{k_l-m-1} \begin{pmatrix} k_l-m-1 \\ k_l-m-i \end{pmatrix} g_i G_l^{(k_l-m-i)}(-\beta_l) + g_{k_l-m} G_l(-\beta_l), \tag{4.16}$$

where

$$g_p \;=\; (-1)^p (p-1)! \sum_{a=1, a\neq l}^{K} \frac{k_a}{(\beta_a-\beta_l)^p} \quad \text{for } p \geq 1.$$

By looking at Equation (4.16) and using an inductive argument, it can be realized that

$$G_l^{(k_l-m)}(-\beta_l) \;=\; G_l(-\beta_l) \times \tilde{G}_l^{(k_l-m)}(-\beta_l), \text{ where} \tag{4.17}$$

$$\tilde{G}_l^{(k_l-m)}(-\beta_l) = \sum_{i=1}^{k_l-m-1} \begin{pmatrix} k_l-m-1 \\ \\ k_l-m-i \end{pmatrix} g_i \tilde{G}_l^{(k_l-m-i)}(-\beta_l) + g_{k_l-m}. \tag{4.18}$$

Now, using Equations 4.14 and 4.17, $C_{l,m}$ can be expressed as

$$C_{l,m} = \frac{G_l(-\beta_l) \times \tilde{G}_l^{(k_l-m)}(-\beta_l)}{(k_l-m)!}$$

Accordingly,

$$\overline{F}_{AR(t)}(b \mid n, \mathbf{k}) = \sum_{l=1}^{x} \prod_{a=1,a\neq l}^{K} b_a^{-k_a} \times G_l(-\beta_l) \sum_{m=1}^{k_l} \frac{1}{b_l^{(k_l-m)}} \times \frac{\tilde{G}_l^{(k_l-m)}(-\beta_l)}{(k_l-m)!} \times F_l(n,m).$$

Since $\beta_i = (1/b_i)$, using Equation (4.15), we can write

$$G_l(-\beta_l) = \prod_{a=1,a\neq l}^{K} \left(\frac{b_a b_l}{b_l - b_a}\right)^{k_a}.$$

Finally,

$$\overline{F}_{AR(t)}(b \mid n, \mathbf{k}) = \sum_{l=1}^{x} \prod_{a=1,a\neq l}^{K} \left(\frac{b_l}{b_l - b_a}\right)^{k_a} \sum_{m=1}^{k_l} \frac{1}{b_l^{(k_l-m)}} \times \frac{\tilde{G}_l^{(k_l-m)}(-\beta_l)}{(k_l-m)!} \times F_l(n,m).$$

In the remainder of this section, we present three lemmas. Lemmas 4.3.1 and 4.3.3 give a recursive expressions to compute

$$H_l(m) = \frac{1}{b_l^{(k_l-m)}} \times \frac{\tilde{G}_l^{(k_l-m)}(-\beta_l)}{(k_l-m)!}$$

and $F_l(n,m)$, respectively. Whereas Lemma 4.3.2 forms an intermediate step necessary to prove Lemma 4.3.3. Based on these results, we can then present an algorithm to compute $\overline{F}_{AR(t)}(b \mid n, \mathbf{k})$.

4.3.2 Computation of $H_l(m)$

Lemma 4.3.1 *For* $1 \leq m \leq k_l - 1$,

$$H_l(m) = \frac{1}{k_l - m} \left[\sum_{i=1}^{k_l-m-1} g_i^{''} H_l(m+i) + g_{k_l-m}^{''} \right],$$

where

$$g_i^{''} = (-1)^i \sum_{a=1,a\neq l}^{K} k_a \left(\frac{b_a}{b_l - b_a}\right)^i,$$

and for $m = k_l$, $H_l(m) = 1$.

Proof: For $m = k_l$, it is obvious that $H_l(m) = 1$. For $1 \leq m \leq k_l - 1$, consider Equation (4.18, of the chapter), then we can write

$$H_l(m) = \frac{1}{b_l^{(k_l-m)}} \times \frac{1}{(k_l-m)!}\left[\sum_{i=1}^{k_l-m-1}\begin{pmatrix} k_l-m-1 \\ k_l-m-i \end{pmatrix} g_i \tilde{G}_l^{(k_l-m-i)}(-\beta_l) + g_{k_l-m}\right]$$

$$= \frac{1}{(k_l-m)!}\left[\left[\begin{pmatrix} k_l-m-1 \\ k_l-m-1 \end{pmatrix} \times \frac{g_1}{b_l} \times \frac{\tilde{G}_l^{(k_l-m-1)}(-\beta_l)}{b_l^{k_l-m-1}} + \begin{pmatrix} k_l-m-1 \\ k_l-m-2 \end{pmatrix} \times \right.\right.$$
$$\frac{g_2}{b_l^2} \times \frac{\tilde{G}_l^{(k_l-m-2)}(-\beta_l)}{b_l^{k_l-m-2}} + \begin{pmatrix} k_l-m-1 \\ k_l-m-3 \end{pmatrix} \times \frac{g_3}{b_l^3} \times \frac{\tilde{G}_l^{(k_l-m-3)}(-\beta_l)}{b_l^{k_l-m-3}} + \cdots + \tag{4.19}$$
$$\begin{pmatrix} k_l-m-1 \\ 2 \end{pmatrix} \times \frac{g_{k_l-m-2}}{b_l^{(k_l-m-2)}} \times \frac{\tilde{G}_l^{(2)}}{b_l^2} + \begin{pmatrix} k_l-m-1 \\ 1 \end{pmatrix} \times \frac{g_{k_l-m-1}}{b_l^{(k_l-m-1)}} \times \frac{\tilde{G}_l^{(1)}}{b_l}$$
$$\left. + \begin{pmatrix} k_l-m-1 \\ 0 \end{pmatrix} \times \frac{g_{k_l-m}}{b_l^{(k_l-m)}}\right]$$

Note,

$$g_i = (-1)^i (i-1)! \sum_{a=1,a\neq l}^{K} k_a \left(\frac{b_a b_l}{b_l - b_a}\right)^i, \quad \text{for } i \geq 1.$$

Let

$$g_i = (-1)^i (i-1)! \sum_{a=1,a\neq l}^{K} k_a \left(\frac{b_a}{b_l - b_a}\right)^i, \quad \text{for } i \geq 1.$$

Accordingly, Equation 4.19 can be written as

$$= \frac{1}{(k_l-m)}\left[\left[\begin{pmatrix} k_l-m-1 \\ k_l-m-1 \end{pmatrix} \times g_1^{'} \times \left(\frac{1}{b_l^{(k_l-m-1)}} \times \frac{\tilde{G}_l^{(k_l-m-1)}(-\beta_l)}{(k_l-m-1)!}\right) + \right.\right.$$

$$\begin{pmatrix} k_l-m-1 \\ k_l-m-2 \end{pmatrix} \times g_2^{'} \times \left(\frac{1}{b_l^{(k_l-m-2)}} \times \frac{\tilde{G}_l^{(k_l-m-2)}(-\beta_l)}{(k_l-m-1)!}\right) + \begin{pmatrix} k_l-m-1 \\ k_l-m-3 \end{pmatrix}$$

$$\times g_3^{'} \times \left(\frac{1}{b_l^{(k_l-m-3)}} \times \frac{\tilde{G}_l^{(k_l-m-3)}(-\beta_l)}{(k_l-m-1)!}\right) + \cdots + \begin{pmatrix} k_l-m-1 \\ 2 \end{pmatrix} \times g_{k_l-m-2}^{'} \times$$

$$\left(\frac{1}{b_l^2} \times \frac{\tilde{G}_l^{(2)}(-\beta_l)}{(k_l-m-1)!}\right) + \begin{pmatrix} k_l-m-1 \\ 1 \end{pmatrix} \times g_{k_l-m-1}^{'} \times \left(\frac{1}{b_l} \times \frac{\tilde{G}_l^{(1)}(-\beta_l)}{(k_l-m-1)!}\right)$$

$$\left. + \begin{pmatrix} k_l-m-1 \\ 0 \end{pmatrix} \times \frac{g_{k_l-m}^{'}}{(k_l-m-1)!}\right].$$

Solving combinatorial and factorial terms, we get

$$\begin{aligned} = \quad & \frac{1}{(k_l-m)}\left[\frac{g_1^{'}}{0!} \times \left(\frac{1}{b_l^{(k_l-m-1)}} \times \frac{\tilde{G}_l^{(k_l-m-1)}(-\beta_l)}{(k_l-m-1)!}\right) + \frac{g_2^{'}}{1!} \times \left(\frac{1}{b_l^{(k_l-m-2)}}\right.\right. \\ & \left.\times \frac{\tilde{G}_l^{(k_l-m-2)}(-\beta_l)}{(k_l-m-2)!}\right) + \frac{g_3^{'}}{2!} \times \left(\frac{1}{b_l^{(k_l-m-3)}} \times \frac{\tilde{G}_l^{(k_l-m-3)}(-\beta_l)}{(k_l-m-3)!}\right) + \\ & \cdots + \frac{g_{k_l-m-2}^{'}}{(k_l-m-3)!} \times \left(\frac{1}{b_l^2} \times \frac{\tilde{G}_l^{(2)}(-\beta_l)}{2!}\right) + \frac{g_{k_l-m-1}^{'}}{(k_l-m-2)!} \times \left(\frac{1}{b_l}\right. \\ & \left.\left.\times \frac{\tilde{G}_l^{(1)}(-\beta_l)}{1!}\right) + \frac{g_{k_l-m}^{'}}{(k_l-m-1)!}\right]. \end{aligned}$$

Let

$$g_1^{''} \;=\; (-1)^i \sum_{a=1, a\neq i}^{K} k_a \left(\frac{b_a}{b_l-b_a}\right)^i.$$

Accordingly,

$$\begin{aligned} = \quad & \frac{1}{(k_l-m)}\left[g_1^{''} \times \left(\frac{1}{b_l^{(k_l-m-1)}} \times \frac{\tilde{G}_l^{(k_l-m-1)}(-\beta_l)}{(k_l-m-1)!}\right) + g_2^{''} \times \left(\frac{1}{b_l^{(k_l-m-2)}} \times\right.\right. \\ & \left.\frac{\tilde{G}_l^{(k_l-m-2)}(-\beta_l)}{(k_l-m-2)!}\right) + g_3^{''} \times \left(\frac{1}{b_l^{(k_l-m-3)}} \times \frac{\tilde{G}_l^{(k_l-m-3)}(-\beta_l)}{(k_l-m-3)!}\right) + \cdots + \\ & \left. g_{k_l-m-2}^{''} \times \left(\frac{1}{b_l^2} \times \frac{\tilde{G}_l^{(2)}(-\beta_l)}{2!}\right) + g_{k_l-m-1}^{''} \times \left(\frac{1}{b_l} \times \frac{\tilde{G}_l^{(1)}(-\beta_l)}{1!}\right) + g_{k_l-m}^{''}\right], \end{aligned}$$

which implies that for $1 \leq m \leq k_l - 1$,

$$H_l(m) = \frac{1}{k_l - m}\left[\sum_{i=1}^{k_l-m-1} g_i^{''} H_l(m+i) + g_{k_l-m}^{''}\right]. \square$$

4.3.3 Computation of $F_l(n,m)$

To compute $F_l(n,m)$, we first analyze the expression

$$\frac{H[m, m-n, m+1, (b/b_l)]}{m\ B(m, n-m+1)},$$

denoted as $R_l(n,m)$ in the following discussion.

Since $1 \leq m \leq n$, the second term $(m-n)$ of the hyper-geometric function $H[m, m-n, m+1, (b/b_l)]$ either gets a value of 0 or a negative integer. Therefore, the summation representing the hyper-geometric function (see 4.10) gets truncated after $(n-m)$ terms. Based on this observation and the fact that

$$\frac{1}{m\ B(m, n-m+1)} = \begin{pmatrix} n \\ \\ m \end{pmatrix},$$

we can express $R_l(n,m)$ as

$$R_l(n,m) = \begin{pmatrix} n \\ \\ m \end{pmatrix} + \sum_{u=1}^{n-m} \frac{m(m+1)\cdots(m+u-1)}{u!} \times$$

$$\frac{(m-n)(m-n+1)\ldots(m-n+u)}{(m+1)(m+2)\ldots(m+u)} \left(\frac{b}{b_l}\right)^u \begin{pmatrix} n \\ \\ m \end{pmatrix}$$

$$= \begin{pmatrix} n \\ \\ m \end{pmatrix} + \left[\frac{m \times (m-n)}{1!\ (m+1)} \left(\frac{b}{b_l}\right)^1 + \frac{m \times (m-n)(m-n+1)}{2!\ (m+2)} \left(\frac{b}{b_l}\right)^2 + \cdots + \right.$$

$$\frac{m \times (m-n)(m-n+1)\cdots(-1)}{(n-m)!\ (n)} \left(\frac{b}{b_l}\right)^{n-m} \Bigg] \begin{pmatrix} n \\ m \end{pmatrix}$$

$$= \begin{pmatrix} n \\ m \end{pmatrix} + \left[\frac{(-1)m \times (n-m)}{1!\ (m+1)} \left(\frac{b}{b_l}\right)^1 + \frac{(-1)^2 m \times (n-m)(n-m-1)}{2!\ (m+2)} \left(\frac{b}{b_l}\right)^2 \right.$$

$$\left. + \cdots + \frac{(-1)^{n-m} m \times (n-m)(n-m-1)\cdots(1)}{(n-m)!\ (n)} \left(\frac{b}{b_l}\right)^{n-m} \right] \begin{pmatrix} n \\ m \end{pmatrix}$$

$$= \begin{pmatrix} n \\ m \end{pmatrix} + \left[\frac{(-1)m \times n(n-1)\cdots(n-m)}{!1\ (m+1)!} \left(\frac{b}{b_l}\right)^1 + \frac{(-1)^2 m \times m(m+1)}{2!} \times \right.$$

$$\left. \frac{n(n-1)\cdots(n-m+1)}{(m+2)!} \left(\frac{b}{b_l}\right)^2 + \cdots + \frac{(-1)^{n-m} n!}{(n-m)!\ (m-1)!} \left(\frac{b}{b_l}\right)^{n-m} \right]$$

$$R_l(m,n) = \begin{pmatrix} n \\ m \end{pmatrix} + \left[(-1) \begin{pmatrix} m \\ 1 \end{pmatrix} \begin{pmatrix} n \\ m+1 \end{pmatrix} \left(\frac{b}{b_l}\right)^1 + (-1)^2 \begin{pmatrix} m+1 \\ 2 \end{pmatrix} \right.$$

$$\left. \begin{pmatrix} n \\ m+2 \end{pmatrix} \left(\frac{b}{b_l}\right)^2 + \cdots + (-1)^{n-m} \begin{pmatrix} n-1 \\ n-m \end{pmatrix} \begin{pmatrix} n \\ n \end{pmatrix} \left(\frac{b}{b_l}\right)^{n-m} \right]. \tag{4.20}$$

Using Equation 4.20, now we present a lemma which will be used in formalizing the recursive expression for $F_l(n,m)$.

Lemma 4.3.2 *Let $y = (t/b_l)$. For $2 \leq m \leq n$,*

$$R_l(n,m) = \frac{(-1)^m}{(m-1)!} \frac{d}{dy}^{(m-1)} \left(\frac{F_l(n,1)-1}{y}\right).$$

Proof: By induction on m.

Basic Step. Consider $m = 2$. From Equation (4.20),

$$R_l(n,2) \;=\; \begin{pmatrix} n \\ 2 \end{pmatrix} + (-1)2 \begin{pmatrix} n \\ 3 \end{pmatrix} y + (-1)^2 3 \begin{pmatrix} n \\ 4 \end{pmatrix} y^2 + \cdots + (-1)^{n-2}(n-1) \begin{pmatrix} n \\ n \end{pmatrix} y^{n-2}. \tag{4.21}$$

From combinatorial theory, we know

$$(1-y)^n \;=\; \sum_{r=0}^{n} \begin{pmatrix} n \\ r \end{pmatrix} y^r, \tag{4.22}$$

which implies that

$$\frac{(1-y)^n - 1}{y} \;=\; (-1) \begin{pmatrix} n \\ 1 \end{pmatrix} + (-1)^2 \begin{pmatrix} n \\ 2 \end{pmatrix} y + (-1)^3 \begin{pmatrix} n \\ 3 \end{pmatrix} y^2 + \cdots + (-1)^n \begin{pmatrix} n \\ n \end{pmatrix} y^{n-1}$$

Differentiate both sides with respect to y,

$$\begin{aligned} \frac{d}{dy}\left[\frac{(1-y)^n - 1}{y}\right] &= 0 + (-1)^2 \begin{pmatrix} n \\ 2 \end{pmatrix} + (-1)^3 2 \begin{pmatrix} n \\ 3 \end{pmatrix} y + (-1)^4 3 \begin{pmatrix} n \\ 4 \end{pmatrix} y^2 \\ &= + \cdots + (-1)^n (n-1) \begin{pmatrix} n \\ n \end{pmatrix} y^{n-2} \end{aligned}$$

$$= \begin{pmatrix} n \\ 2 \end{pmatrix} + (-1)2 \begin{pmatrix} n \\ 3 \end{pmatrix} y + (-1)^2 3 \begin{pmatrix} n \\ 4 \end{pmatrix} y^2 + \cdots + (-1)^{n-2}(n-1) \begin{pmatrix} n \\ n \end{pmatrix} y^{n-2}. \tag{4.23}$$

From Equations (4.21) and (4.23),

$$R_l(n,2) \;=\; \frac{d}{dy}\left[\frac{F_l(n,1) - 1}{y}\right].$$

Induction Hypothesis. Assume that for some $3 \leq i < n$,

$$R_l(n,i) \;=\; = \;\frac{(-1)^i}{(i-1)!} \frac{d}{dy}^{(i-1)} \left(\frac{F_l(n,1) - 1}{y}\right).$$

Induction Step. Using Equation (4.20), we can write

$$
\begin{aligned}
R_l(n,i) &= \begin{pmatrix} n \\ i \end{pmatrix} + (-1)i \begin{pmatrix} n \\ i+1 \end{pmatrix} y + (-1)^2 \frac{(i+1)i}{2!} \begin{pmatrix} n \\ i+2 \end{pmatrix} y^2 + (-1)^3(i+2) \\
&\quad \frac{(i+1)i}{3!} \begin{pmatrix} n \\ i+3 \end{pmatrix} y^3 + \cdots + (-1)^{n-i} \frac{(n-1)\cdots(i+1)i}{(n-i)!} \begin{pmatrix} n \\ n \end{pmatrix} y^{n-i},
\end{aligned}
$$

and

$$
\begin{aligned}
R_l(n,i+1) &= \begin{pmatrix} n \\ i+1 \end{pmatrix} + (-1)(i+1) \begin{pmatrix} n \\ i+2 \end{pmatrix} y + (-1)^2 \frac{(i+2)(i+1)}{2!} \begin{pmatrix} n \\ i+3 \end{pmatrix} \\
&\quad y^2 + (-1)^3 \frac{(i+3)(i+2)(i+1)}{3!} \begin{pmatrix} n \\ i+4 \end{pmatrix} y^3 + \cdots + (-1)^{n-i-1}(n-1) \\
&\quad \frac{(n-2)\cdots(i+1)}{(n-i-1)!} \begin{pmatrix} n \\ n \end{pmatrix} y^{n-i-1}.
\end{aligned}
$$

Realizing from these two equations,

$$
R_l(n,i+1) = \frac{(-1)}{i}\frac{d}{dy}[R_l(n,i)] = \frac{(-1)}{i}\frac{(-1)^i}{(i-1)!}\frac{d}{dy}\left(\frac{d}{dy}^{(i-1)}\left(\frac{F_l(n,1)-1}{y}\right)\right).
$$

This proves,

$$
R_l(n,i+1) = \frac{(i-1)^{i+1}}{i!}\frac{d}{dy}^{(i)}\left(\frac{F_l(n,1)-1}{y}\right). \ \square
$$

Using Lemma 4.3.2, we prove another lemma which gives us an efficient and numerically stable recursive formalism to compute $F_l(n,m)$.

Lemma 4.3.3 *Let* $y = (b/b_l)$*. For* $m = 1$,

$$
F_l(n,m) = (1-y)^n,
$$

and for $2 \leq m \leq n$*,*

$$F_l(n,m) = \begin{pmatrix} n \\ m-1 \end{pmatrix} y^{m-1} (1-y)^{n-(m-1)} + F_l(n,m-1).$$

Proof: This lemma is proved in two parts. The first part proves the lemma for $m = 1$, while the second part proves it for $2 \leq m \leq n$.

(1) For $m = 1$**:** Using Equation 4.20,

$$R_l(n,1) = \begin{pmatrix} n \\ 1 \end{pmatrix} + (-1)\begin{pmatrix} n \\ 2 \end{pmatrix} y + (-1)^2 \begin{pmatrix} n \\ 3 \end{pmatrix} y^2 + \cdots + (-1)^{n-1} \begin{pmatrix} n \\ n \end{pmatrix} y^{n-1}.$$

From Equation (4.13), we know that

$$\begin{aligned} F_l(n,1) &= 1 - yR_l(n,1), \\ &= 1 + (-1)\begin{pmatrix} n \\ 1 \end{pmatrix} y + (-1)^2 \begin{pmatrix} n \\ 2 \end{pmatrix} y^2 + \cdots + (-1)^n \begin{pmatrix} n \\ n \end{pmatrix} y^n. \end{aligned}$$

Using Equation 4.22, we can finally write

$$F_l(n,1) = (1-y)^n \ . \ \square$$

(2) For $2 \leq m \leq n$: By induction on m.

Basic Step. Let $m = 2$. Using Lemmas 4 and 5,

$$R_l(n,2) = \frac{1}{1!}\frac{d}{dy}\left[\frac{(1-y)^n - 1}{y}\right] = \frac{1 - ny(1-y)^{n-1} - (1-y)^n}{y^2}.$$

Using Equation (4.13),

$$F_l(n,2) = 1 - y^2 R_l(n,2),$$

which proves

$$F_l(n,2) \ = \ ny(1-y)^{n-1} + F_l(n,1).$$

Induction Hypothesis. Assume that for $3 \leq i < n$,

$$F_l(n,i) \ = \ \binom{n}{i-1} y^{i-1}(1-y)^{n-(i-1)} + F_l(n,i-1).$$

Induction Step. Using Lemma 5,

$$R_l(n,i+1) \ = \ \frac{(-1)}{i}\frac{d}{dy}[R_l(n,i)].$$

Also, it is clear from Lemma 5 that $R_l(n,i)$ has a negative sign for odd values of i and a positive sign for even values of i. Keeping this fact in mind and using Equation (4.13), $R_l(n,i)$ can be expressed as

$$R_l(n,i) \ = \ (-1)^i \left(\frac{F_l(n,i)-1}{y^i}\right).$$

Suppose i is odd, then

$$R_l(n,i+1) = \frac{(-1)}{i}\frac{d}{dy}\left[(-1)\frac{F_l(n,i)-1}{y^i}\right] \ = \ \frac{1}{i}\frac{d}{dy}\left[\frac{F_l(n,i)-1}{y^i}\right].$$

From induction hypothesis,

$$F_l \ = \ \sum_{k=0}^{i-1} \binom{n}{k} y^k(1-y)^{n-k}.$$

Accordingly,

$$R_l(n,i+1) \ = \ \frac{1}{i}\frac{d}{dy}\left[\frac{\sum_{k=0}^{i-1}\binom{n}{k} y^k(1-y)^{n-k} \ - \ 1}{y^i}\right]$$

$$= \frac{-\begin{pmatrix} n \\ i \end{pmatrix} y^i(1-y)^{n-i} + 1 - \sum_{k=0}^{i-1} \begin{pmatrix} n \\ k \end{pmatrix} y^k(1-y)^{n-k}}{y^{i+1}}.$$

$$F_l(n, i+1) \;=\; 1 - y^{i+1} R_l(n, i+1) \;=\; \begin{pmatrix} n \\ i \end{pmatrix} y^i(1-y)^{n-i} + \sum_{k=0}^{i-1} \begin{pmatrix} n \\ k \end{pmatrix} y^k(1-y)^{n-k},$$

which gives,

$$F_l(n, i+1) \;=\; \begin{pmatrix} n \\ i \end{pmatrix} y^i(1-y)^{n-i} + F_l(n, i).$$

Similarly, when i is even, then

$$R_l(n, i+1) \;=\; \frac{(-1)}{i} \frac{d}{dy} \left[\frac{1 \;-\; \sum_{k=0}^{i-1} \begin{pmatrix} n \\ k \end{pmatrix} y^k(1-y)^{n-k}}{y^i} \right]$$

$$= \;(-1) \left[\frac{\begin{pmatrix} n \\ i \end{pmatrix} y^i(1-y)^{n-i} - 1 + \sum_{k=0}^{i-1} \begin{pmatrix} n \\ k \end{pmatrix} y^k(1-y)^{n-k}}{y^{i+1}} \right].$$

This gives

$$F_l(n, i+1) \;=\; \begin{pmatrix} n \\ i \end{pmatrix} y^i(1-y)^{n-i} + F_n(n, i). \;\Box$$

In the next section, using results from Lemmas 4.3.1 and 4.3.3, we present an efficient and numerically stable algorithm to compute $\overline{F}_{AR(t)}(b \mid n, \mathbf{k})$.

4.4 Algorithm to Compute $\overline{F}_{AR(t)}(b \mid n, \mathbf{k})$

Recall that we can express the complementary conditional distribution in terms of $H_l(m)$ and $F_l(n,m)$ as

$$\overline{F}_{AR(t)}(b \mid n, \mathbf{k}) = \sum_{l=1}^{x} \prod_{a=1, a\neq l}^{K} \left(\frac{b_l}{b_l - b_a} \right)^{k_a} \sum_{m=1}^{k_l} H_l(m) \times F_l(n,m).$$

While the recursions developed in the previous section give us an efficient method to calculate $H_l(m)$ and $F_l(n,m)$, $H_l(m)$ will take on extremely large positive and negative values for certain $\mathbf{k}$ and large n. These values can cause loss of precision in a practical implementation of the algorithm. This loss of precision can be avoided, to some extent, by writing a recursion for $H_l(m) \times F_l(n,m)$ directly, which we denote by $X_l(n,m)$. In this notation,

$$\overline{F}_{AR(t)}(b \mid n, \mathbf{k}) = \sum_{l=1}^{x} \prod_{a=1, a\neq l}^{K} \left(\frac{b_l}{b_l - b_a} \right)^{k_a} \sum_{m=1}^{k_l} X_l(n,m). \tag{4.24}$$

Then, using Lemma (2), we can write a recursive expression to compute $X_l(n,m)$. Since $\frac{X_l(n,m)}{F_l(n,m)} = H_l(m)$, Lemma 2 says that

$$\frac{X_l(n,m)}{F_l(n,m)} = \frac{1}{(k_l - m)} \sum_{i=1}^{k_l - m - 1} g_i'' \times \frac{X_l(n,m+i)}{F_l(n,m+i)} + g_{k_l - m}'',$$

for $1 \leq m \leq k_l - 1$. This implies that

$$X_l(n,m) = \frac{1}{(k_l - m)} \sum_{i=1}^{k_l - m - 1} g_i'' \times X_l(n,m+i) \times \frac{F_l(n,m)}{F_l(n,m+i)} + g_{k_l - m}'' \times F_l(n,m). \tag{4.25}$$

Note for $m = k_l$, $X_l(n,m) = F_l(n,m)$, since $H_l(k_l) = 1$. By writing the recursion this way, we avoid the problem with large $H_l(m)$, since the $F_l(n,m)$ are probabilities.

The formulation presented in (4.24) and (4.25) diminishes the numerical difficulty in computing $\overline{F}_{AR(t)}(b \mid n, \mathbf{k})$, but still requires careful implementation to avoid loss of precision in the computation. In particular, the formulation calls for the addition of a large number of positive and negative numbers, which will be large in magnitude for certain $\mathbf{k}$ and large n. These numbers may be very close to one another, since when added together they should equal a potentially very small probability. The main loss of precision in such cases is caused by the subtraction of two numbers which are close in value. In this case, a large shift to the left will be made after the operation to normalize the result, and many digits of precision will be lost.

Keeping this in mind, to minimize the number of subtractions, we split $X_l(n,m)$ into positive $X_l^{pos}(n,m) > 0$ and negative $X_l^{neg}(n,m) < 0$ parts such that

$$X_l(n,m) \;=\; X_l^{pos}(n,m) + X_l^{neg}(n,m).$$

Similar to the recursion in Equation 4.25, for $1 \le m \le k_b - 1$, recursions for $X_l^{pos}(n,m)$ and $X_l^{neg}(n,m)$ can be expressed as

$$\begin{aligned} X_l^{pos}(n,m) \;=\; & \frac{1}{(k_l-m)}\left[\sum_{i=1}^{k_l-m-1} I(g_i^{''} > 0) g_i^{''} X_l^{pos}(n,m+i)\frac{F_l(n,m)}{F_l(n,m+i)} + \right. \\ & \left. \sum_{i=1}^{k_l-m-1} I(g_i^{''} < 0) g_i^{''} X_l^{neg}(n,m+i)\frac{F_l(n,m)}{F_l(n,m+i)} \;+\; I(g_{k_l-m}^{''} > 0) g_{k_l-m}^{''}\right] \end{aligned} \tag{4.26}$$

and

$$\begin{aligned} X_l^{neg}(n,m) \;=\; & \frac{1}{(k_l-m)}\left[\sum_{i=1}^{k_l-m-1} I(g_i^{''} > 0) g_i^{''} X_l^{neg}(n,m+i)\frac{F_l(n,m)}{F_l(n,m+i)} + \right. \\ & \left. \sum_{i=1}^{k_l-m-1} I(g_i^{''} < 0) g_i^{''} X_l^{pos}(n,m+i)\frac{F_l(n,m)}{F_l(n,m+i)} \;+\; I(g_{k_l-m}^{''} < 0) g_{k_l-m}^{''}\right], \end{aligned} \tag{4.27}$$

where $I(x)$ is an indicator function (returns 1 if x is true and else 0) and for $m = k_l$, $X_l^{pos}(n,m) = F_l(n,m)$ and $X_l^{neg}(n,m) = 0$.

Given that all the additions are performed in ascending order of absolute magnitudes, then, for each shifted reward $b_l > b$, Equations 4.26 and 4.27 yield an accurate (to the precision of the computation) positive value and negative value. Both these values are then multiplied by

$$\prod_{a=1,a\neq l}^{K} \left(\frac{b_l}{b_l - b_a}\right)^{k_a},$$

which is denoted z in the algorithm below. Then, these two resulting numbers with powers greater than 0 are subtracted to yield a probability. Since the subtraction will result in a probability, the order of magnitude of the two numbers will always be the same (called *magnitude* in the following algorithm). This implies that to obtain an answer accurate to some specified precision, we need a decimal-digit precision in the computation equal to the sum of *magnitude* and the desired precision. If normal floating point math (as per the IEEE standard) does not provide the required precision, the algorithm obtains the required precision by invoking multi-precision routines, rolling back, and recomputing the required terms and sums. Thus, by the use of selective multi-precision, the algorithm always achieves the desired precision and, hence, is numerically stable. Since there is only a single subtraction needed for each shifted rate reward, each summation will at most roll back once, leading to an efficient algorithm.

More precisely, the algorithm to compute $\overline{F}_{AR(t)}(b \mid n, \mathbf{k})$ is as follows. (Note that the given algorithm does not consider impulse rewards. However, an extension of the algorithm to include impulse rewards is straightforward and can be done in a manner similar to one presented by Qureshi and Sanders [51].)

Algorithm 4.4.1 *(Compute the complementary conditional distribution, $\overline{F}_{AR(t)}(b \mid n, \mathbf{k})$, using (4.24) and (4.25).)*

$\overline{F}_{AR(t)}(b \mid n, \mathbf{k}) \;=\; 0.$

For $l = 1$ *to* x

For $m = 1$ *to* k_l

Compute $F_l(n, m)$ *by using recursion given in Lemma (3).*

End(For).

For $m = k_l$ *to* 1

If $(m \ = \ k_l)$

$X_l^{pos}(n, k_l) \ = \ F_l(n, k_l)$, *and* $X_l^{neg}(n, k_l) \ = \ 0.$

Else

Use Equation 4.26 to compute $X_l^{pos}(n, m)$.

Use Equation 4.27 to compute $X_l^{neg}(n, m)$.

End(For).

Let $X_{negative}$ *be the sum of all* $X_l^{neg}(n, m)$, *added in the ascending order of their absolute magnitudes.*

Let $X_{positive}$ *be the sum of all* $X_l^{pos}(n, m)$, *added in the ascending order of their magnitudes.*

$X_{positive} \ = \ z \times X_{positive}.$

$$X_{negative} \;=\; z \times X_{negative}.$$

Let magnitude be the power (in base 10) of $X_{positive}$ *and* $X_{negative}$.

Let needed digits = magnitude + desired precision.

If (IEEE floating point precision > needed digits)

$$X_{total} \;=\; X_{positive} + X_{negative}$$

Else

Recompute X_{total} *by recomputing* l_{th} *iteration*

of for loop with needed digits precision.

$$\overline{F}_{AR(t)}(b \mid n, \mathbf{k}) \;=\; \overline{F}_{AR(t)}(b \mid n, \mathbf{k}) + X_{total}.$$

End(For)

This algorithm, together with the path-based approach described in Section 4.2, provide the basis for stable and efficient calculation of the distribution of time-averaged reward accumulated over a finite interval. The overall algorithm to compute the distribution of accumulated reward starts by first generating possible paths and computing their respective probabilities. The probabilities of paths which correspond to identical $\mathbf{k}$ vectors are added together, while the probabilities of paths corresponding to distinct $\mathbf{k}$ vectors are stored separately. This exploration of paths and computation of path probabilities continues until either all paths with probabilities greater than or equal to the discarding weight are explored or the memory used reaches a preset machine-dependent limit. After reaching one of these conditions, the conditional distribution of accumulated reward is computed for each $\mathbf{k}$ vector generated, and after being multiplied by the probability of the explored paths corresponding the the $\mathbf{k}$ vector, is added together with the results from other paths to form the unconditional distribution. If all paths with probability greater than the threshold have not yet been explored, the process is repeated, exploring more paths and then computing the required conditional distributions.

To test the algorithm on a non-trivial example, we have implemented it in C++, using the multi-precision library LiDIA [52]. The implementation has been used for several large examples and works well when the number of paths required to compute a desired accuracy is not more than tens of millions. One such example, with over one-half million states is given in the next section.

4.5 Example Study

The applicability of our approach is illustrated by considering the performability analysis of the highly redundant fault-tolerant system depicted in Figure 3.1. Using SANs, we modeled the system with three non-repairable computers. (Note that our algorithm applies to repairable, as well as non-repairable systems.) The Markov-level representation of the model is very large, consisting of 463,268 states even if reduced base model construction [5] is used to detect symmetries in the process. To the best of our knowledge, this is by far the largest and most complicated model that has ever been solved for the distribution of accumulated reward, and it illustrates the practicality of our approach.

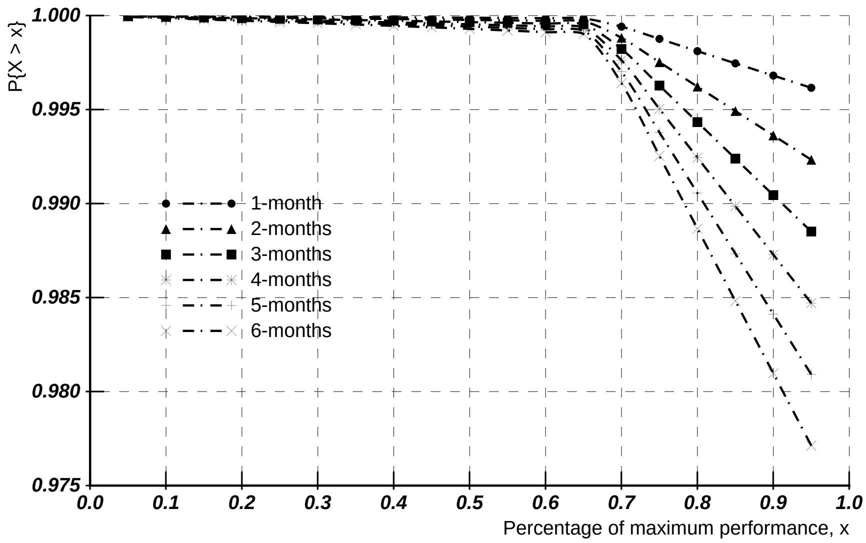

Figure 4.1: Performability distribution

The performability measure considered is the complementary PDF of the fraction of jobs completed in the interval, considering faults, relative to the number jobs that would complete in a fault-free system. More specifically, we consider a system where 300 jobs complete per second if 3 processors are working, 200 jobs complete per second if 2 processors are working, and 100 jobs complete per second if 1 processor is working. The measure, then, is the PDF of the number of jobs that complete divided by $300 * T$, where T is the length of the interval under consideration.

We present results regarding both the performability of the considered system and the efficiency of the algorithm in the context of the example. In particular, Figure 4.1 depicts the complementary distribution of reward accumulated during time intervals of one to six months. Each line on the graph corresponds to a specific time interval, and it gives the probability that the system performs at a level higher than the value given on the x-axis.

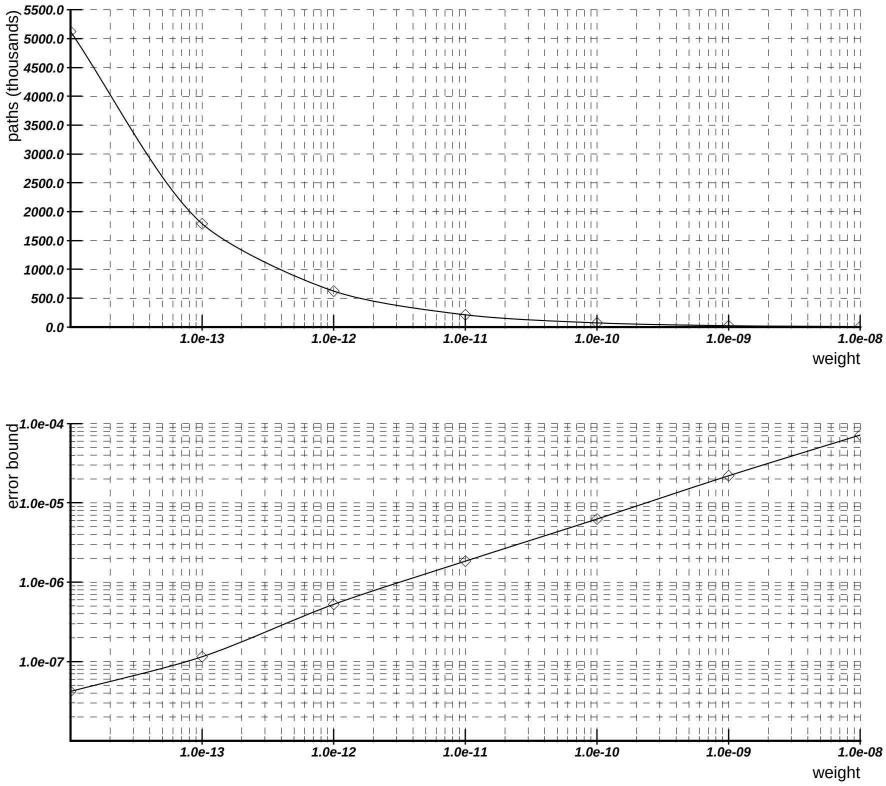

Figure 4.2: Effect of discarding of paths (t = one year)

A value of $w = 10^{-11}$ was used for all time intervals, resulting in an accuracy of more than six digits for each result.

Figure 4.2 illustrates the effect of changing w on the number of paths considered and the bound on the error obtained due to discarding paths whose probabilities are less than the value w. For this example, we consider a utilization period of one year. Since the space required by previous approaches depends critically on the length of the interval, this (reasonable) time point would result in an unreasonably large memory requirement for current non-path-based approaches. Note that the number of paths that need to be considered decreases dramatically with an increasing weight, while the error bound achieved remains reasonable. This illustrates the effectiveness of our approach on systems with up to millions of probable paths.

CHAPTER 5

Application

To illustrate the usefulness of stochastic activity networks in modeling real systems, we use them to evaluate the performance and availability of voting algorithms, taking into account the effect of workload. In particular, we want to illustrate that 1) SANs can indeed be used to represent system characteristics (such as workload, performance, and dependability) and details of voting algorithms in a single model, and 2) it is possible to solve the model to collect interesting information regarding performance and availability of voting algorithms.

The next section summarizes the importance of voting algorithms and, in light of the previous work, argues the significance of our study of voting algorithms.

5.1 Overview

Dependable access to data is critically important to many distributed system applications. This dependability is most often achieved by replication of data on multiple nodes of a system. In this way, the failure of some subset of the nodes can be tolerated, maintaining the availability of data to a user of the system. However, increased availability often comes at a cost in performance, since a user must be provided with a consistent view of the data in spite of possible network and node failures. By consistent, we mean that the user is guaranteed to see the most current replicated copy.

Replica control algorithms have been developed to provide data consistency. Many replica control algorithms are based on the idea of voting, which was first presented by Thomas [55] and later generalized by Gifford [56]. In such schemes, each node is assigned a vote v_i. Read and write quorums (numbers of votes of r and w, respectively) are then chosen such that only one write operation can be performed at any time and no read operations can be performed while a write operation is being performed. This behavior can be achieved by requiring that $r + w$ be greater than the total number of votes assigned to the nodes and w be greater than half of the total number of votes assigned to the nodes.

Algorithms which assign a fixed number of votes to each node are called "static" voting algorithms. After Gifford, several variants of the basic static algorithm were introduced to improve availability and performance (e.g., see [57, 58, 59, 60]). Algorithms which assign a dynamic number of votes to each host, depending on the state of the system, have also been considered. Dynamic algorithms can increase the availability of data, since votes can be reassigned after a component failure to maintain the desired behavior (single concurrent write and no reads while a write is being performed) among the remaining functioning hosts (e.g., [61, 62, 63, 64, 65]).

Many variants of voting algorithms have been proposed, and a large number of these variants have been evaluated, either via simulation, simple probabilistic arguments, or state-space-based methods (e.g., Markov models, stochastic Petri nets). With a few exceptions, however, the evaluations have focused on determining the availability or reliability of particular algorithms, as determined by the specific algorithm and failure rates of particular system components, or the "cost" associated with implementing the voting algorithm. While these studies provided useful information concerning the subject protocols, they were limited in several respects.

First, by basing the criteria for proper operation (i.e., an available system) on the structural configuration of the system, previous evaluations do not provide a user-oriented view of availability. In particular, such evaluations judge a system as unavailable if the resources to form a quorum are not present, regardless of whether a request to form a quorum has been made. Thus, the service delivered by a system would be considered improper even if no request for the service had been made. This view is contrary to that of the dependable computing community ([66]), which bases the proper/improper service criteria on the ability of a system to deliver proper service *when it is requested.* Given this view, there is growing evidence (both experimental and model-based) that workload can affect fault-tolerant system dependability (e.g., see [67, 68, 69, 70]). It is therefore reasonable to assume that workload is also an important determinant of availability in replicated data systems and should be considered.

Second, most studies have been limited to studying the impact of a particular protocol and network on the availability of the data, without simultaneously considering the performance delivered by the system. Failures of system components can significantly affect the performance of a replicated data system as perceived by a user. In many applications, performance measures such as the mean response time can be useful in determining the suitability of particular voting algorithms. This fact has been recognized by several researchers (e.g., see [71, 72, 73, 74]), which underscores the importance of determining the performance as well as availability of voting algorithms.

Stochastic Petri nets can be used to evaluate the performance and availability of voting algorithms in a single model, taking into account the effect of workload. In particular, due to the abstract nature of their model primitives, stochastic Petri nets (e.g., DSPNs, GSPNs, SANs, and SRNs) can be used to represent system characteristics (such as workload, performance, and dependability) and details of voting algorithms in a single model. Stochastic Petri nets were first applied to the evaluation of voting algorithms by Dugan and Ciardo [75], who evaluated the effect of changing the number of copies on data availability. They did not, however, study the effect of workload on availability or determine the performance of the algorithm in the presence of faults.

In this chapter, we use SANs to evaluate the performance and availability of particular static and dynamic voting algorithms, taking into account the effects of workload and component failures. Both server and network failures are considered. Our measure of performance is the mean response time of a write request to the system, accounting for the fact that a request may have to wait until the number of copies required to form a quorum become available. Among other things, we show that the mean response time of the voting algorithms considered decreases with an increase in the number of copies employed and an *increase* in workload. The decrease in mean response time with increasing workload is nonintuitive and is linked to the particular voting algorithms we considered.

5.2 Network Architecture, Voting Algorithms, and Workload Model

5.2.1 Network Architecture Model

We consider a networked LAN environment, where a single broadcast-capable link, i.e., "network," connects the hosts together. A host on the network behaves as either a client or a server. Each server carries a replicated data object. Clients generate requests for the replicated data, while the servers service the requests.

Furthermore, we assume that the number of replicated data objects is equal to the number of servers and each replicated data object resides on a different server. A data object becomes unavailable if either its server or the network fails. For simplicity, we assume that each replicated data object consists of a single file copy and all the data objects carry the copy of the same file. Since each server carries a replicated copy of the file, it is sufficient to monitor a server's status to keep track of the status of the file copy.

Servers and the network fail independently of each other and can fail at any time other than when the replicated file system is in reconstruction. Since reconstruction time is very short relative to the failure rates of the network and servers, it is reasonable to assume that failures do not occur during reconstruction. We assume that failure and repair times are exponentially distributed, with fixed rates. We also assume that a network failure prevents all communication on the LAN. In this case, the replicated file system becomes unavailable and cannot provide any service (including completing service of a request in the system). As soon as the network is repaired, it becomes available and once again begins processing requests, if conditions are such that it can form a quorum.

Repair starts as soon as a server or the network fails. We take a conservative approach, as suggested in [75], and always consider a repaired server as outdated. How outdated a repaired copy is depends upon the number of write requests it has missed. However, in order to model the replicated file system with a reasonably small state space, we treat all outdated copies similarly and ignore the degree to which they are outdated.

5.2.2 Voting Algorithms

Many voting algorithms could have been used for this study. We consider one static and one dynamic voting algorithm similar to those considered by Jajodia and Mutchler [63]. Since they only differ in the requirement for quorum formation, we describe the static voting algorithm and highlight the differences as necessary.

The protocols make use of two types of version numbers associated with each copy of data, one "physical" and the other "logical." The *physical* version number is an indication of the version of the data at the server. The *logical* version number is the version that was written during the last quorum in which the copy participated. Note that the physical version number may be less than the logical version number for a copy; this indicates that the copy participated in a quorum formation but (as is possible in the protocol) did not have its data updated. Given these two version numbers, there are three classes into which a copy can fall.

The first class is where both the physical and logical version numbers indicate that the copy participated in the most recent quorum formation. These copies are called *physically current* copies. The second class is where the logical version number indicates that the copy participated in the most recent quorum, but the physical version number is less than

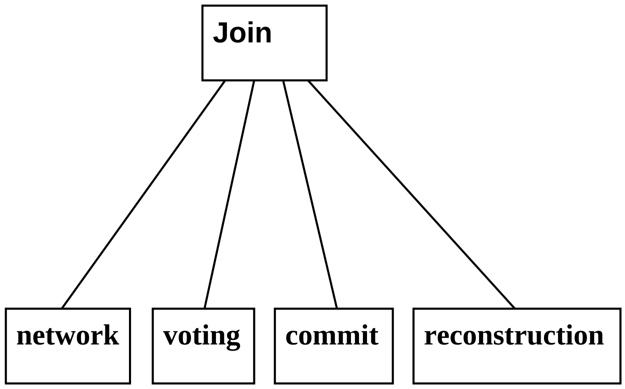

Figure 5.1: Composed model: *file system model*

the logical version number, indicating that the copy is out of date relative to the most recent update. We call these copies *logically current* copies. The third and final case is when the logical version number indicates that the copy did not participate in the last quorum formation. In this case, the data of the copy are also outdated, since the physical version number must be less than or equal to the logical version number. We call these copies *outdated*. Both the static and dynamic algorithms specify that a write request can be executed if there exists at least one physically current copy and the number of physically or logically current copies participating in quorum formation is greater than half of some number X. The selection of this number X is what differentiates the static and dynamic algorithms. The static algorithm specifies that X is equal to the total number of copies, while the dynamic algorithm requires that it must be equal to the number of copies that took part in the service of the last request.

More precisely, each file f has an associated version number VN which is initially set to 0 and then incremented with each write request on the file. Each copy f_i of the file f has a physical version number PN_i to keep track of the physical status of the copy and logical version number LN_i to keep track of the last quorum that copy participated in. A copy f_i

of the file f is said to be physically current if $PN_i = VN$. It is logically current but not physically current if $LN_i = VN$ but $PN_i < VN$. Otherwise, it is outdated. Furthermore, each copy f_i has an associated variable SC_i to keep track of the number of copies that took part in the service of the last request. (Note that variable SC_i is only required in dynamic voting.) Under the normal operation (when a quorum can be formed), the two phases *voting* and *commit* are executed in response to a write request.

Voting After generating a write request for a file f, the client enters the voting phase. In this phase, the client transmits messages to the servers carrying copies of the file. Each server S_i locks the local copy f_i and sends the physical and logical version numbers PN_i and LN_i back to the client. (Note that in the case of dynamic voting each server also sends variable SC_i back to the client.) After receiving messages from the servers, the client first determines the greatest logical version number among the responses it received. It then checks to see whether it received a number of responses with this logical version number greater than half of the number of copies either in the system for static voting or that took part in the service of the last request for dynamic voting. It then checks to see if at least one of the copies with $LN_i = VN$ has $PN_i = VN$. If so, then the client has obtained a physically current copy and can proceed to do the update and commit the write. If not, the client sends messages to the servers to release their locks, and the file system is forced into reconstruction.

Commit If a quorum could be formed during the voting phase, the system enters the commit phase. In this phase, the client increments the VN associated with the file f and sends the updates to all servers that responded to the request for a quorum formation. Each receiving server S_i updates its local copy f_i, makes the associated $PN_i = LN_i = VN$, and releases the lock. Also, in the case of dynamic voting, each server updates the variable SC_i equal to the number of copies that took part in the quorum formation before releasing the lock. Note that in the variant of the algorithm studied in this chapter, we physically update all servers that responded to the last quorum formation request. We do this because we assume a broadcast-capable network where all servers can be updated at a cost equal to that of updating a single server. When the cost to physically update multiple servers is greater than a single server (as it is on a nonbroadcast-capable network, for example), it may not be beneficial to update all servers; the algorithm itself only requires that we increment the logical version number of more than half the servers and physically update at least one server.

Reconstruction is only invoked if the system is unable to form a quorum in the voting phase. All servers, including failed servers just repaired, are physically updated during the reconstruction phase. The write request which causes the invocation of reconstruction waits during the reconstruction phase and triggers voting again once it is complete.

5.2.3 Workload Model

The workload model captures the demands placed on the data. In a real system, requests for the data may come from many sources and result in contention for the data. The file system can only service one write request at a time, so requests must be queued, and serviced sequentially. Since our goal is to understand the effect of such a workload on the protocol, it suffices to model the workload as a simple generator of requests, with an exponentially distributed time between the completion of one request and the receipt of the next. This point of view simplifies the model, as there is no need to represent the clients

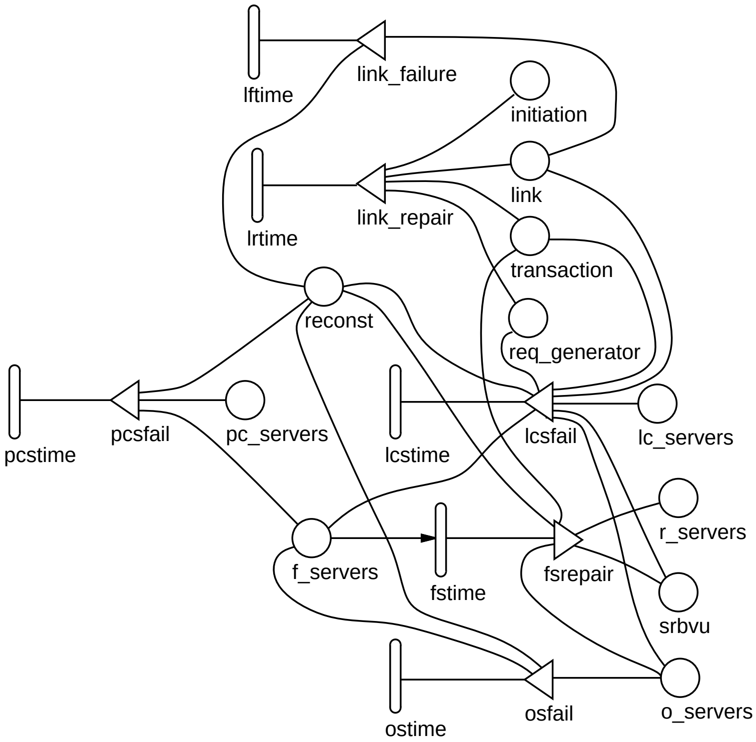

Figure 5.2: SAN model: *network*

individually. Furthermore, since the workload model insures that there is no more than one write request at a time in the system, we do not explicitly need to model the locking mechanism which insures that there is at most one write request in the system.

5.3 Stochastic Activity Network Representation

We use SANs, a stochastic extension of Petri nets, to represent each replicated file system model. As argued in the introduction, SANs provide a convenient mechanism for representing algorithms and hardware, such as is considered here, and can be automatically converted into Markov processes, which are then numerically solved. Space does not permit a review of the theory of SANs, but interested readers are encouraged to consult [2, 39].

To aid in their understanding, the static and dynamic replicated file system models are divided into four submodels: *network*, *voting*, *commit*, and *reconstruction*, representing the file system architecture, the workload and the voting phase, the commit phase, and the reconstruction process, respectively. The four submodels are then joined by connecting certain of their places together to generate a complete model of the replicated file system. In SAN terminology, this model is referred to as a *composed model* [5] and is given in Figure 5.1. The remainder of the section describes the four SAN submodels in detail, illustrating how the static algorithm, hardware, and workload described previously are represented as SANs. During the discussion, we also highlight the minimal changes which are required to represent the dynamic algorithm.

5.3.1 Network

Figure 5.2 is a SAN model of the network architecture of the replicated file system. It specifies precisely how server and network failures and repairs occur and the effect these failures and repairs have on the replicated file system. Tables 5.1, 5.2, and 5.3 define the components of the SAN. As depicted in Figure 5.2, the markings of places *pc_servers*, *lc_servers*, *o_servers*, and *f_servers* represent the numbers of physically current servers, logically current servers, outdated servers, and failed servers, respectively. Initially, the marking of place *pc_servers* is set to the number of servers (copies) in the system, while the markings of places *lc_servers*, *o_servers*, and *f_servers* are set to zero. Place *r_servers* is a temporary holding place for servers and represents the number of servers that are repaired while the system is in reconstruction. Place *link* represents the status of the network. As discussed in the previous section, the network is nondegradable, i.e., it either is functioning and can deliver messages or has failed and cannot provide communication. A marking of one in this place indicates the network is functioning; a marking of zero indicates that the network has failed. Four different failure situations need to be considered: failure of a physically current server, failure of a logically current server, failure of an outdated server, and failure of the network. The failure of servers and the network is modeled using input gates and timed activities. In particular, activity *pcstime* and input gate *pcsfail* represent the failures of physically current servers. Input gate *pcsfail* specifies the enabling condition for the timed activity *pcstime*. In particular, as shown in Table 5.1, activity *pcstime* is enabled when there is at least one physically current server and the system is not in reconstruction. All timed activities are assigned exponential rates with per hour units.

Activity *lcstime* and input gate *lcsfail* represent the failures of logically current servers, in a manner similar to that of physically current servers. However, since we allow servers to fail during the commit phase, these failures must be handled in a manner somewhat different from the failure of physically current servers. This complication need not be considered for physically current servers, since all physically current servers become logically current at the beginning of the commit phase (they have not yet been made physically current by the current write transaction) and become physically current at the end of the commit phase, since the network has broadcast capability. Servers can fail during the commit phase, however, and, if they do, cannot be made physically current. Input gate *lcsfail* thus specifies that the failure of all remaining logically current servers during the commit phase ends the phase by setting the marking of place *transaction* to zero. If the network is available, another request is now allowed, and a token is placed in place *req_generator* to

Table 5.1: Input gate definitions for SAN model *network*

Gate	*Definition*
pcsfail	Predicate MARK(reconst) == 0 && MARK(pc_servers) > 0
	Function MARK(pc_servers) − = 1; MARK(f_servers) + = 1;
link_failure	Predicate MARK(link) == 1 && MARK(reconst) == 0
	Function MARK(link) = 0;
link_repair	Predicate MARK(link) == 0
	Function MARK(link) = 1; if (MARK(initiation) == 0 && MARK(req_generator) == 0 && MARK(transaction) == 0) { MARK(req_generator) = 1; }
osfail	Predicate MARK(reconst) == 0 && MARK(o_servers) > 0
	Function MARK(o_servers) − = 1; MARK(f_servers) + = 1;
lcsfail	Predicate MARK(reconst) == 0 && MARK(lc_servers) > 0
	Function MARK(lc_servers) − = 1; if (MARK(transaction) == 1 && MARK(lc_servers) == 0) { MARK(srbvu) + = 1; MARK(o_servers) + = (MARK(srbvu) + 1); MARK(srbvu) = 0; MARK(transaction) = 0; if (MARK(link) == 1) MARK(req_generator) = 1; } else MARK(f_servers) + = 1;

Table 5.2: Activity time distributions for SAN model *network*

Activity	*Distribution*	*Rate*
pcstime	exponential	0.01∗MARK(pc_servers)
fstime	exponential	1∗MARK(f_servers)
lftime	exponential	0.002
lrtime	exponential	2
ostime	exponential	0.01∗MARK(o_servers)
lcstime	exponential	0.01∗MARK(lc_servers)

Table 5.3: Output gate definitions for SAN model *network*

Gate	*Definition*
fsrepair	*if (MARK(reconst) == 1) {* *MARK(r_servers) + = 1; }* *else {* *if (MARK(transaction) == 1) {* *MARK(srbvu) + = 1; }* *else {* *MARK(o_servers) + = 1; } }*

indicate that it should be generated (see the SAN workload representation discussed in the next section).

The failure of the network is represented by activity *lftime* and input gate *link_failure*. As specified in Table 5.2, we suppose that network failures are exponentially distributed with a rate of 0.002 per hour. Failure of the network will block the completion of service of an ongoing transaction and hence negatively impact its response time.

Server and network repairs begin as soon the respective component has failed and are modeled by activities *fstime* and *lrtime*, respectively. Output gate *fsrepair* specifies what happens when a server is repaired (upon completion of activity *fstime*). Specifically, if a server is repaired during the reconstruction phase (i.e., the marking of place *reconst* is one), it joins the set of repaired servers that need to made physically current (i.e., the marking of place *r_servers* is incremented). If instead the server is repaired during the commit phase, then it is temporarily held in the place *srbvu* so that it can be distinguished from the available servers which formed the quorum in the voting phase and hence are not made physically current. Finally, if the server was repaired when the system was not in the

Table 5.4: Input gate definitions for SAN model *voting*

Gate	*Definition*
voting	Predicate $MARK(initiation)$ == 1 && $MARK(link)$ == 1
	Function $MARK(initiation) = 0;$ *if* ($MARK(pc\_servers) > 0$ && $(MARK(pc\_servers) + MARK(lc\_servers)) >$ $(MARK(o\_servers) + MARK(f\_servers)))$ { $MARK(transaction) = 1;$ $MARK(lc\_servers) += MARK(pc\_servers);$ $MARK(pc\_servers) = 0;$ $MARK(lc\_servers) += MARK(o\_servers);$ $MARK(o\_servers) = 0;$ } *else* { $MARK(reconst) = 1;$ $MARK(r\_servers) += MARK(pc\_servers);$ $MARK(pc\_servers) = 0;$ $MARK(r\_servers) += MARK(lc\_servers);$ $MARK(lc\_servers) = 0;$ $MARK(r\_servers) += MARK(o\_servers);$ $MARK(o\_servers) = 0;$ }

reconstruction or commit phase, it joins the set of outdated servers (marking of *o_servers* is incremented).

Activity *lrtime* specifies that the time to repair the network is exponentially distributed and has a repair rate of two per hour. Input gate *link_repair* specifies what happens when the network is repaired. If a transaction was blocked due to the failed link, it can now continue.

5.3.2 Workload and Voting

Figure 5.3 shows the SAN model of the workload and voting phase of the algorithm. Table 5.4 gives the definitions of the components of the SAN. We first consider how the workload is implemented using SAN components. Recall that as specified in the previous section, we model the time between completion of one request and the initiation of the next as an exponentially distributed random variable. Activity *gen_time* represents this exponential time and is enabled whenever place *req_generator* contains a token. (The initial marking of *req_generator* is one.) Completion of *gen_time* removes the token from *req_generator*, disabling the start of service of a second request. Completion of the request returns the token to place *req_generator*, once again enabling activity *gen_time*. The rate of activity *gen_time* is varied to see the effect of workload on system availability and mean response time. Note that by disabling activity *gen_time* during the processing of a request, we have implicitly insured that at most one request will be serviced by the system at a time, as is required by the protocol.

Completion of *gen_time* places a token in place *initiation* and signifies that the voting phase of the algorithm should begin. At this point, the client attempts to form a quorum by broadcasting the request to all servers. As shown in Table 5.4, quorum formation will be attempted if there is a request to do so ($MARK(initiation)$==1) and the network is able to provide communication ($MARK(link)$ ==1). The time required to broadcast a message to all the servers asking them to participate in the quorum and to receive replies from the nonfailed servers is represented by activity *comm_time*. The rate associated with the timed activity *comm_time* (45000000/($MARK(pc\_servers)$+$MARK$ ($lc\_servers$)+$MARK$ ($o\_servers$)+1.0)) is calculated by considering a 10 Mbps (megabits per second) network and a mean message size (exponentially distributed) of 100 bytes for both read and reply messages. After the client receives the reply messages from the nonfailed servers, the function associated with gate *voting* is executed. If the number of servers

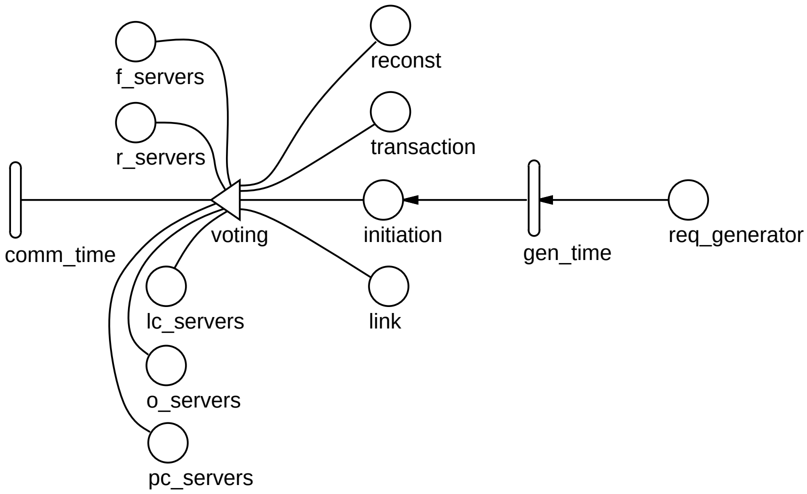

Figure 5.3: SAN model: *voting*

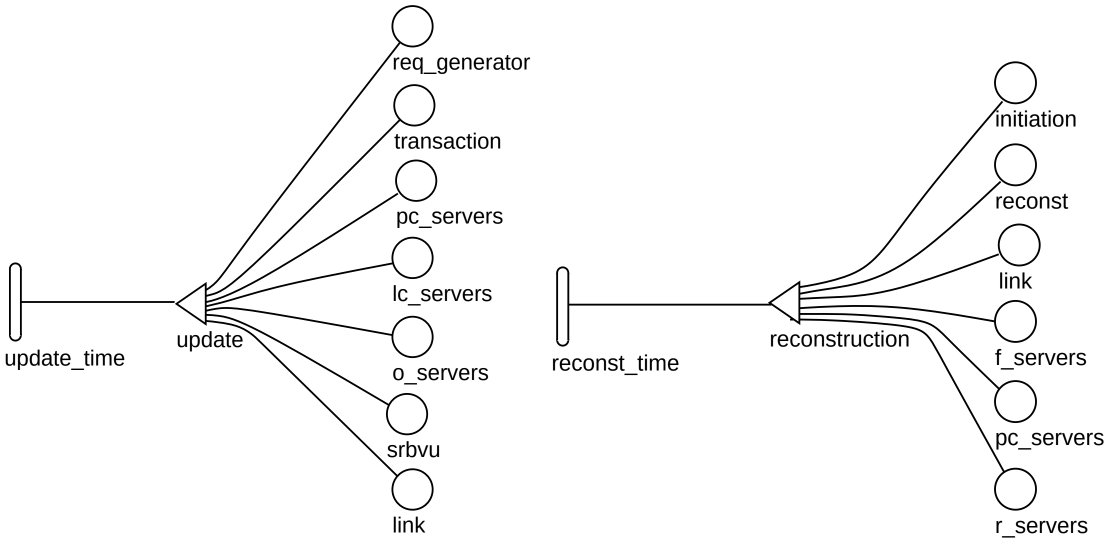

Figure 5.4: SAN models: *commit* and *reconstruction*

Table 5.5: Input gate definitions for SAN model *commit*

Gate	*Definition*
update	Predicate $MARK(transaction) == 1 \ \&\& \ MARK(link) == 1$
	Function $MARK(pc\_servers) \ += MARK(lc\_servers);$ $MARK(lc\_servers) = 0;$ $MARK(o\_servers) \ += MARK(srbvu);$ $MARK(srbvu) = 0;$ $MARK(transaction) = 0;$ $MARK(req\_generator) = 1;$

which are logically current is greater than half the total number of servers and at least one of these servers is physically current, then the attempt to form a quorum is a success in static voting. However, for dynamic voting, the number of logically current servers is required to be greater than half of the number of servers that took part in servicing the last request. To do this, an additional place is added to the SAN in Figure 5.3 to keep track of the number of servers that took part in servicing the last request. Regardless of the type of algorithm, in the case of success, the client progresses to the commit phase of the algorithm. Otherwise, the file system is forced into reconstruction. In both situations, the token is removed from the place *initiation*. The commit phase is triggered by placing a token in place *transaction*, while the reconstruction phase is initiated by placing a token in the place *reconst*.

5.3.3 Commit

The left SAN in Figure 5.4 and Table 5.5 model the commit phase of the algorithm. The commit phase of the algorithm is entered if a quorum was successfully formed during the voting phase. In this case, all physically current servers become logically current, since the write request in the system has not yet been executed. Placement of a token in place *transaction* during the voting phase signifies that this indeed is the case and enables activity *update_time*. The time associated with the activity *update_time* corresponds to the time to broadcast the update message to all servers that responded to the request to form a quorum and the time required to write the update to the disk at all such servers. Since we consider a network with broadcast capability, the time associated with the activity *update_time* does not depend on the number of such servers and can be calculated by adding a single expected transmission time and disk write time. The rate assigned to *update_time* $(3600/(1000 * (8/10^7 + 1/(2 \times 10^6))))$ corresponds to a 10 Mbps network, a 2 MBps (megabytes per second) disk, and a mean update message size of 1000 bytes.

The completion of the timed activity *update_time* signifies the completion of the commit phase and triggers the execution of the function associated with the input gate *update*. As can be seen from Table 5.5, the function first updates the logically current servers then changes the status of the servers which were repaired during the update phase (and collected in place *srbvu*) to *outdated*. Recall that during the commit phase, the SAN collects the repaired servers in place *srbvu* so that they can be distinguished from the available servers which took part in voting. The marking of the place *transaction* is then set to zero, and the activity representing the time between requests to the replicated file system is enabled by placing a token in the place *req_generator*.

5.3.4 Reconstruction

The right SAN in Figure 5.4 and Table 5.6 illustrate the model for the reconstruction phase. Recall, as discussed in the previous section, that the system will be forced into

Table 5.6: Input gate definitions for SAN model *reconstruction*

Gate	*Definition*
reconstruction	Predicate $MARK(reconst) == 1$ && $MARK(f\_servers) == 0$ && $MARK(link) == 1$
	Function $MARK(pc\_servers) += MARK(r\_servers);$ $MARK(r\_servers) = 0;$ $MARK(reconst) = 0;$ $MARK(initiation) = 1;$

reconstruction if a quorum cannot be formed during the voting phase. This action is represented in the SAN by having input gate *voting* (in the submodel *voting*) place a token in place *reconst*. When this occurs, the algorithm specifies that all failed servers are to be repaired and then all servers are to be made physically current. Input gate *reconstruction* thus specifies that activity *reconst_time* not be enabled until all the failed servers are repaired. Furthermore, in the case of dynamic voting, the marking of the place, which keeps track of the number of servers which participated in servicing the last request, is set to the number of copies in the system.

After they are repaired, *reconst_time* is enabled and specifies the reconstruction time. When reconstruction is completed (*reconst_time* completes), all servers are marked as physically current, and a token is placed in place *initiation* (to initiate another round of voting) so that the write request which forced reconstruction can now be serviced.

5.4 Results

Given the SAN models described in the previous section, the stochastic process representations for the static and dynamic models are automatically constructed using *UltraSAN*

[39]. The construction procedure results in a continuous-time, discrete-state ergodic Markov process for each set of input parameter values. Depending on the number of servers considered, the state-space sizes for the generated models range from 66 (3 copies) to 375 (9 copies) states and from 120 (3 copies) to 2640 (9 copies) states, respectively. The resulting Markov processes can then be solved (via *UltraSAN*) using successive over-relaxation. Solution time is quick, on the order of a few seconds, so we can evaluate the algorithm for a wide variety of parameter values. In particular, we determine the availability and mean response time of the system to write requests when:

1. The mean time between the completion of one request and the arrival of the next takes on values from .0833 to 30 minutes (denoted as "Average Time" in Figure 5.5).
2. The number of copies (i.e., servers) takes odd values from 3 to 9.

Availability can be determined directly from the model by measuring the fraction of time the system is not in reconstruction. Mean response time is determined by calculating the average number of requests in the system (since at most one request can be in the system, the average number of requests in the system is equivalent to the probability that there is one request in the system), the average arrival rate of requests to the system, and applying Little's result.

5.4.1 Static Voting

The left graph in Figure 5.5 depicts the effect of the change in workload and number of copies on availability. It shows that either an increase in the number of copies or a decrease in workload or both increases availability.

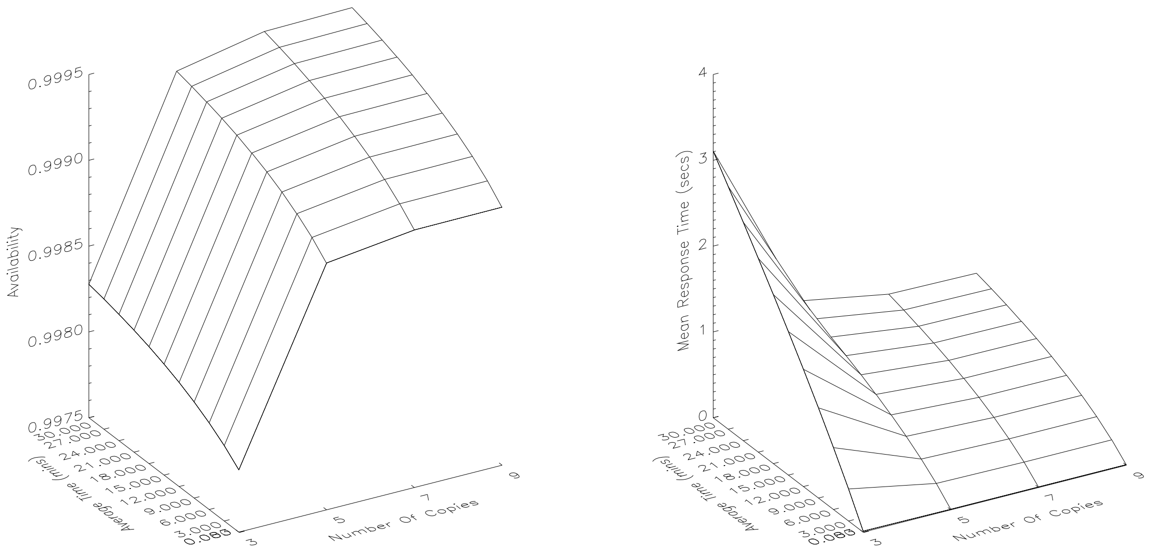

Figure 5.5: Effect of increasing the number of copies and decreasing workload on availability and response time using static voting

We consider first the increase in availability as the number of copies is increased for a fixed workload. This trend is well understood and has been reported in the context of many voting algorithms. To understand it in terms of the SAN model, consider the different structural configurations the system can be in. By *configuration*, we mean a distribution of servers among physically current, logically current, outdated, and failed classes. Each configuration can be categorized as proper or improper, depending on whether a quorum can be formed in the configuration. Since each server carries a single copy of the replicated file, an increase in the number of copies implies an increase in the number of servers. Each additional server demands more server failures before the system enters an improper configuration. As shown by the model, the number of proper configurations increases, and, furthermore, the probability of an incoming request finding the system in a proper configuration increases. Availability thus increases with an increase in the number of copies.

Looking at the graph, one can see that the increase in availability is significant when the number of copies is increased from 3 to 5 but becomes negligible when the number of copies is increased more than 5. Specifically, for a mean time of 0.0833 minutes between the completion of one request and the arrival of the next, the increase in availability is 1.08×10^{-3} when the number of copies is increased from 3 to 5, while the increase in availability is on the order of 10^{-5} when the number of copies is increased from 5 to 7 or 7 to 9.

The increase in availability with a decrease in workload is not so well known or intuitive. Recall that the system is available whenever it is not in reconstruction. A decrease in workload could in principle push the availability of the system in two different directions. On one hand, since the system is forced into reconstruction only if a request arrives that cannot be serviced, a decrease in rate of requests would imply that the rate at which events occur which can *possibly* cause the system to go into reconstruction decreases, and hence availability might increase. On the other hand, a longer time between requests implies that it is more likely that a server fails and is repaired between requests, leaving it in an outdated state and unable to help in forming a quorum. Thus, a decrease in workload implies that the probability of entering reconstruction on a per-arrival basis increases (in other words, the "possibility" of reconstruction becomes more probable for each request that occurs). Thus, availability might decrease with decreasing workload.

Which trend dominates depends on the particular workload, failure rates, and repair rates considered. For the range of model parameters considered, the first trend dominates, and, as seen in left graph of Figure 5.5, availability increases with decreasing workload. Specifically, for the failure and repair rates considered, the increase in availability is on the

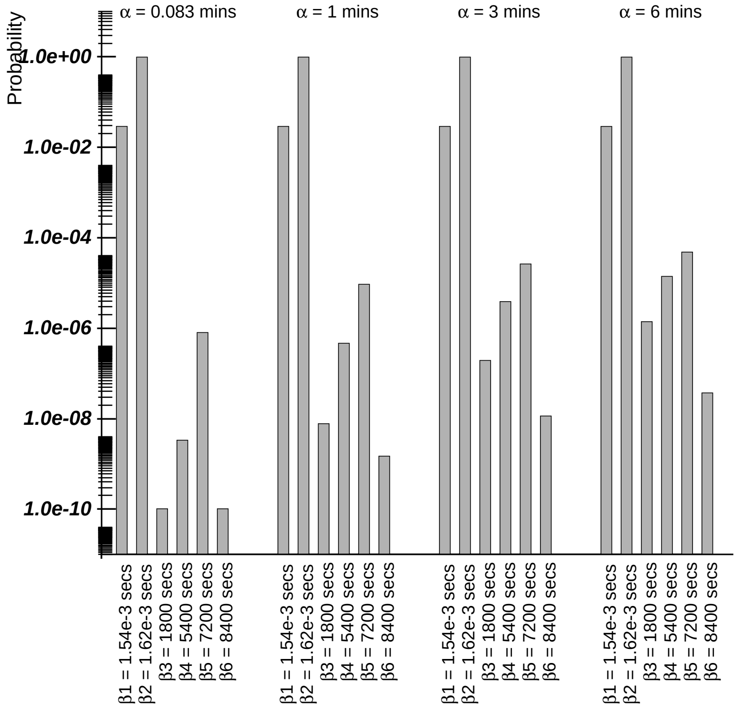

Figure 5.6: Distribution of the probability of being in structural configurations with different conditional mean response times

order of 10^{-4} over the range of workload studied. Furthermore, the increase in availability with the increase in time between requests is slightly more significant for higher numbers of copies. In particular, for an increase in the mean time between the completion of one request and the arrival of the next from 0.0833 to 30 minutes, the increase in availability for the number of copies equal to 3, 5, 7, and 9 is 4.15×10^{-4}, 4.51×10^{-4}, 4.90×10^{-4}, and 4.96×10^{-4}, respectively.

The right graph in Figure 5.5 illustrates that the mean response time decreases with an increase in the number of copies but increases with a decrease in workload. Recall that the mean response time is defined to be the average time it takes the system to respond to a write request, including the time waiting for the system to become available, if it is forced into reconstruction.

The decrease in response time with an increasing number of copies can be attributed to the same mechanism that caused an increase in availability with an increase in the number of copies. Simply put, an increase in the number of copies increases the probability that the file system is in a proper configuration. Since proper configurations have a much shorter conditional mean response time (on the order of milliseconds as compared to thousands of seconds, if reconstruction must be invoked), this increase results in a decrease in mean response time with an increasing number of copies. Note that this argument relies on the fact that the expected response time in proper configurations does not increase with an increasing number of copies. This is true for our scenario (since updates can be done via a broadcast) but would not be true if updates must be done in a serial fashion on the network. In this case, depending on values given to model parameters, mean response time might increase with an increasing number of copies. Further work must be done to determine exactly when this will be the case.

As was the case with availability, the decrease in mean response time is greatest when the number of copies is increased from 3 to 5 and is still considerable when the number of copies is increased from 5 to 7, but the decrease becomes negligible when the number of copies is decreased from 7 to 9. Specifically, for an average time of 3.0 minutes between the completion of one request and the arrival of the next, the decrease in the response time for an increase in the number of copies from 3 to 5, 5 to 7, and 7 to 9 is 0.195 seconds, 0.012 seconds, and 4.82×10^{-4} seconds, respectively.

The right graph in Figure 5.5 also illustrates the nonintuitive result that, for a fixed number of copies, the average response time increases with a decrease in workload. This fact can be understood by considering the second trend discussed when considering the effect of workload on availability: the probability an incoming request finds the system in an improper configuration increases with decreasing workload. In particular, recall that as the mean time between the completion of one request and the arrival of another increases, the probability that a host will become outdated, by failing and being repaired, increases. Higher probabilities of configurations with more outdated hosts lead to higher probabilities of improper configurations. Requests that arrive when the system is in an improper configuration take much longer to be serviced, since the system must first undergo reconstruction.

This relationship can be seen quantitatively, by looking at the probability of seeing different expected response times conditioned on being in particular structural configurations. Figure 5.6 shows this for a system with three replicated copies. On this histogram, the parameter α indicates a specific workload (mean time between completion of one request and an arrival of the next), and the β indicates possible expected response times, conditioned on being in particular structural configurations. Note that it is easy to calculate an expected response time conditioned on being in a structural configuration, since the configuration defines precisely the set of steps the transaction must take to be serviced. Since the expected time to complete each of these steps is known, the total expected response time, conditioned on being in each structural state, can be calculated. $\beta 1$ and $\beta 2$ correspond to configurations where service is proper, while $\beta 3$ to $\beta 6$ correspond to configurations that are improper. The graph shows that the probabilities associated with improper configurations increase slightly with decreasing workload. But even this slight increase in probability significantly affects the (unconditioned) expected response time, since the conditional expected response times within these configurations are so large.

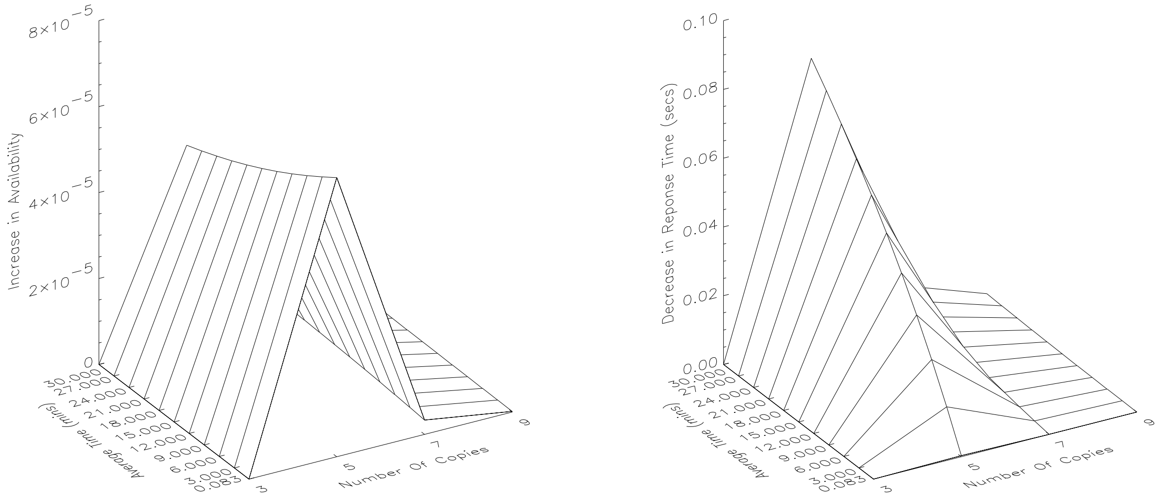

Figure 5.7: Increase in availability and decrease in response time when the static voting is replaced by dynamic voting

5.4.2 Dynamic Voting

The results obtained for dynamic voting suggest that availability and mean response time increase either with an increase in the number of copies or a decrease in the workload. These trends are similar to the one we have discussed for static voting and also stem from the same reasons. However, compared to the results for static voting, the results for dynamic voting show a higher availability and a lower mean response time for a given workload and a fixed number of copies. The reason for higher availability with dynamic voting has already been explained in many previous illustrations. Due to lack of space, we will not elaborate here.

The right graph in Figure 5.7 shows a decrease in mean response time when the static voting algorithm is replaced by the dynamic voting algorithm in a replicated file system. This behavior can be explained by realizing that for a given number of copies certain configurations which are improper for the static voting algorithm become proper for the dynamic voting algorithm. We name such configurations as *dual-behavior* configurations. The static voting invokes reconstruction when an incoming request finds the system in any of the dual-behavior configurations. In contrast, the dynamic voting does not require reconstruction for dual-behavior configurations. As the graph shows, the average response time decreases for number of copies equal to 5, 7, or 9 when the static voting is replaced by the dynamic voting. However, for the number of copies equal to 3, the number of proper configurations remains the same for both static and dynamic voting algorithms. Therefore, there is no change in the response time.

Furthermore, by looking more carefully at the graph in Figure 5.7, one notices that the maximum decrease in the response time is for the number of copies equal to 5. This behavior can be understood by realizing that for higher numbers of copies the probability of finding the system by an incoming request in any of the dual-behavior configurations decreases since the system requires more failures to go into a dual-behavior configuration. Hence, the decrease in response time is smaller for high numbers of copies.

CHAPTER 6

Conclusion

The objective of my research was to develop a general reward structure that enables one to specify a larger set of performance and dependability measures, a solution technique for instant-of-time measures that enables one to solve systems with large state-spaces, and an efficient and numerical stable method to compute probability distribution functions of interval-of-time measures. This chapter summarizes how these three objectives were achieved. Moreover, it summarizes the study of the effect of varying workload on the availability and response time of voting algorithms.

Generation of a state-level reward model representation is sometimes necessary for an efficient solution of reward measures defined on stochastic Petri nets. In Chapter 2, we developed an algorithm to do this for a general (step-based) reward structure defined on a SAN. The reward structure is more general than previously considered reward structures for SPNs and can represent all reward variables that can be defined on the marking behavior of the net. In addition, we presented an algorithm to determine if a SAN is well specified with respect to a step-based reward structure. Being well specified is a necessary and sufficient condition for the SAN to be completely probabilistically specified, and it allows one to construct a state-level reward model from a given SAN and reward structure.

The well-specified check gives one the freedom to probabilistically specify choices that matter, with respect to the system being modeled, using cases, and to *not* specify choices that do not matter but arise because multiple events (whose order does not matter) can happen at the same time. This cleanly separates choices that the modeler cares about, and those he/she does not care about, and allows for full marking dependency on case probabilities. Furthermore, elimination of unstable markings is done on-the-fly, thus reducing the memory requirement to generate a process representation if the SAN has unstable reachable markings, compared to other algorithms. The algorithm has been implemented in *UltraSAN* and applied successfully in a wide variety of application areas.

Chapter 3 presented a solution for computing the state-occupancy probability vector at desired instants in time. From the state-occupancy vector, the expected value and probability distribution function of an instant-of-time measure can be computed. The method is based on exploring paths of the continuous-time Markov process in a depth-first search manner. Given a path, its probability is computed by using a renewal argument. The probabilities of a finite but large set of significant paths are computed to obtain the state-occupancy probability vector with some tolerable error bound. It is shown that indeed the algorithm can be used to obtain the state-occupancy probability vector of non-trivial models. The method is significant in two ways: 1) it does not require the construction of complete state spaces and 2) it does not require the storage of the complete generator matrix in the memory. Therefore, it avoids the memory complexity of algorithms based on matrix powering, ordinary differential methods, and uniformization.

In Chapter 4, we presented a new method for computing the distribution of time-averaged reward accumulated over a finite interval. The method is based on generating paths of the uniformized process which are probable and computing the probability distribution of the accumulated reward conditioned on each path. To make this practical, we developed a new algorithm for computing the conditional distribution of accumulated reward which is computationally efficient and numerically stable. In addition, we presented experimental results to show that the path-based approach is useful when solving Markov reward models with large state spaces and up to tens of millions of probable paths. We also showed that, for the example considered, the number of paths required to obtain a result decreases dramatically as the path-discarding threshold is increased while still obtaining reasonable accuracy.

In Chapter 5, we studied the effect of varying workload on availability and response time of voting algorithms. The results of the study are significant for two reasons. First, they illustrate that it is indeed possible to construct analytic models of voting protocols that account for the effect of workload and protocol parameter values on performance and availability. Second, they provide interesting results regarding the effect of workload on the performance and availability of the particular voting algorithm studied. Regarding the algorithm itself, the results obtained for availability were as expected. The availability increases either with an increase in the number of copies or a decrease in the workload or both. However, the results obtained for average response time were surprising. The average response time decreases with an increase in the number of copies, while it increases with a decrease in the workload.

While the results are interesting in their own right, they also suggest further work that could be done. For example, factors other than workload and number of copies may also affect a voting algorithm's performance and availability. Specifically, if communication on the network is nonbroadcast, then an important factor would be the number of physical updates which are performed within the commit phase. Seemingly, the availability would increase with increasing numbers of updates, but the behavior of mean response time is not intuitive. Interesting performance/availability tradeoffs may exist that could be investigated using a model similar to the one presented in Chapter 5. Finally, the results suggest that comparisons could be made between different network architectures and voting schemes, taking into account the effect of workload on performance and availability.

Appendix A

The *UltraSAN* modeling environment [1]

A.1 Abstract

Model-based evaluation of computer systems and networks is an increasingly important activity. For modeling to be used effectively, software environments are needed that ease model specification, construction, and solution. Easy-to-use, graphical methods for model specification that support solution of families of models with differing parameter values are also needed. Since no model solution technique is ideal for all situations, multiple analysis- and simulation-based solution techniques should be supported. This paper describes *UltraSAN*, one such software environment. The design of *UltraSAN* reflects its two main purposes: to facilitate the evaluation of realistic computer systems and networks, and to provide a test-bed for investigating new modeling techniques. In *UltraSAN*, models are specified using stochastic activity networks, a stochastic variant of Petri nets, using a graphical X-window-based interface that supports large-scale model specification, construction, and solution. Models may be parameterized to reduce the effort required to solve families of models, and a variety of analysis- and simulation-based solution techniques are

[1] This appendix consists of the paper *UltraSAN modeling environment* co-authored with W. H. Sanders, W. D. Obal, and F. K. Widjanarko, Performance Evaluation, vol. 24, pp. 89-115, 1995.

supported. The package has a modular organization that makes it easy to add new construction and solution techniques as they become available. In addition to describing the features, capabilities, and organization of *UltraSAN*, the paper illustrates the use of the package in the solution for the unreliability of a fault-tolerant multiprocessor using two solution techniques.

A.2 Introduction

Model-based performance, dependability, and performability evaluation has become an integral part of the design process. The primary strength of using models rather than a prototype system is the ease in evaluating competing design decisions. Creating a model usually costs less than building a prototype and, in most cases, produces an accurate model that is much less complex than the system itself. This approach has the added benefit that once a model is created it may be saved for future use in evaluating proposed modifications to the system. So although measurement is and will remain an important method of system evaluation, modeling reduces the evaluation effort by focusing attention on the system characteristics having the largest impact on the performance measures of interest.

Due to these merits and the computational complexity of most model solution methods, the development of effective software tool support is an active research area. Many such tools exist and are based on a wide variety of model specification techniques (e.g., queueing networks, Markov processes, object-oriented languages, and stochastic Petri nets). Several surveys have been written on such tools (see, for example, [76, 77, 78]). Stochastic Petri nets (SPNs) and extensions have been the subject of significant recent research resulting in a number of software tools supporting the evaluation of systems represented in this way. These tools include among many DSPNexpress [79], FIGARO [80], GreatSPN [81], METASAN [82], RDPS [83], SPNP [84], SURF-2 [85], TimeNET [86], and *UltraSAN* [87]. Some of these tools have been built to illustrate particular classes of solution algorithms (e.g., DSPNexpress, analytic solution of SPNs with deterministic and exponential timings), while others have had very broad goals, implementing both analytic- and simulation-based solution methods (e.g., GreatSPN, SPNP, SURF-2, and *UltraSAN*).

This paper describes *UltraSAN*. In *UltraSAN*, models are specified using a variant of SPNs known as stochastic activity networks (SANs), and solution techniques include many analytic- and simulation-based approaches. The development of the software was guided by two goals: the desire to create an environment (test-bed) for experimenting with new model construction and solution techniques, and the goal of facilitating large-scale evaluations of realistic systems using both analytic- and simulation-based techniques. As will be seen in the next section, the first goal dictates that the software be constructed in a modular fashion so new construction and solution techniques can be easily added. The second goal dictates that many different solution techniques be available, since no single technique is ideal in all circumstances.

Since the initial release in 1991 [87], many new construction and solution techniques and user interface features have been added. In particular with respect to the user interface, a parameterized model specification method that facilitates solution of models with multiple sets of input parameter values has been developed. Furthermore, three new analytical solvers, as well as a terminating simulation solver based on importance sampling, have been added to the package. The first new analytic solver solves models that have deterministic as well as exponential activities. The two remaining new analytic solvers compute the probability distribution function and mean, respectively, of reward accumulated over a finite interval. The importance sampling simulator provides an efficient solution for models that contain rare events which are significant relative to a measure in question. These enhancements have improved *UltraSAN*'s usefulness to others and have illustrated its effectiveness as a test-bed for new modeling techniques. The package has now been at academic and industrial sites for about four years, and much has been learned from its use.

Our purpose for describing *UltraSAN* is to provide insight into the most important aspects of the package and the reasons particular design decisions were made. Clearly, space does not permit an extended discussion of the theoretical developments made and incorporated in the package. However, reference is given to these developments, and the interested reader is urged to consult the appropriate references.

The remainder of the paper is organized as follows. Section A.3 of this paper gives an overview of the modeling process, as embodied in *UltraSAN*, including a brief review of SANs and an example illustrating the use of SANs in modeling a fault-tolerant multicomputer. The intent here is to provide a high-level overview of the capabilities and features of the package. Sections A.4–A.7 describe in detail each step of the modeling process. These sections provide more insight regarding the design choices made in each step of the process. Results for the example obtained using two different techniques (uniformization and importance-sampling-based simulation) are given in Section A.8. Finally, Section A.9 summarizes the contributions and features of the tool and suggests areas of future research and development.

A.3 Overview

The organization of *UltraSAN* follows directly from the goals of the package. As shown in Figure A.1, the package is constructed in a modular manner. All of the modules shown are coordinated by a central program called the control panel, which manages the interface between the user and the various modules and provides basic consistency checking. In addition, the control panel offers several useful utilities, such as automatic documentation and convenient viewing of output data. The modules communicate via files which have well-defined formats. This configuration makes it easy to add new functionality, thus contributing to the goal of being a test-bed for developing new modeling techniques.

Figure A.1 is divided into four sections, representing the four phases of the modeling process used in *UltraSAN*: *model specification*, *global variable assignment*, *study construction*, and *study solution*. These phases are briefly described here as an introduction to the package, with more detailed descriptions of their functioning and underlying theory given in the following sections.

Model specification

In the model specification phase, SANs are defined graphically using the SAN editor and then composed into a hierarchical model using the composed model editor. The performance variable editor is used to define the performance measures of interest in terms of a reward structure at the SAN level. Once a model is specified, it is typically solved many times for various values of the input parameters. To accommodate this activity, *UltraSAN* allows

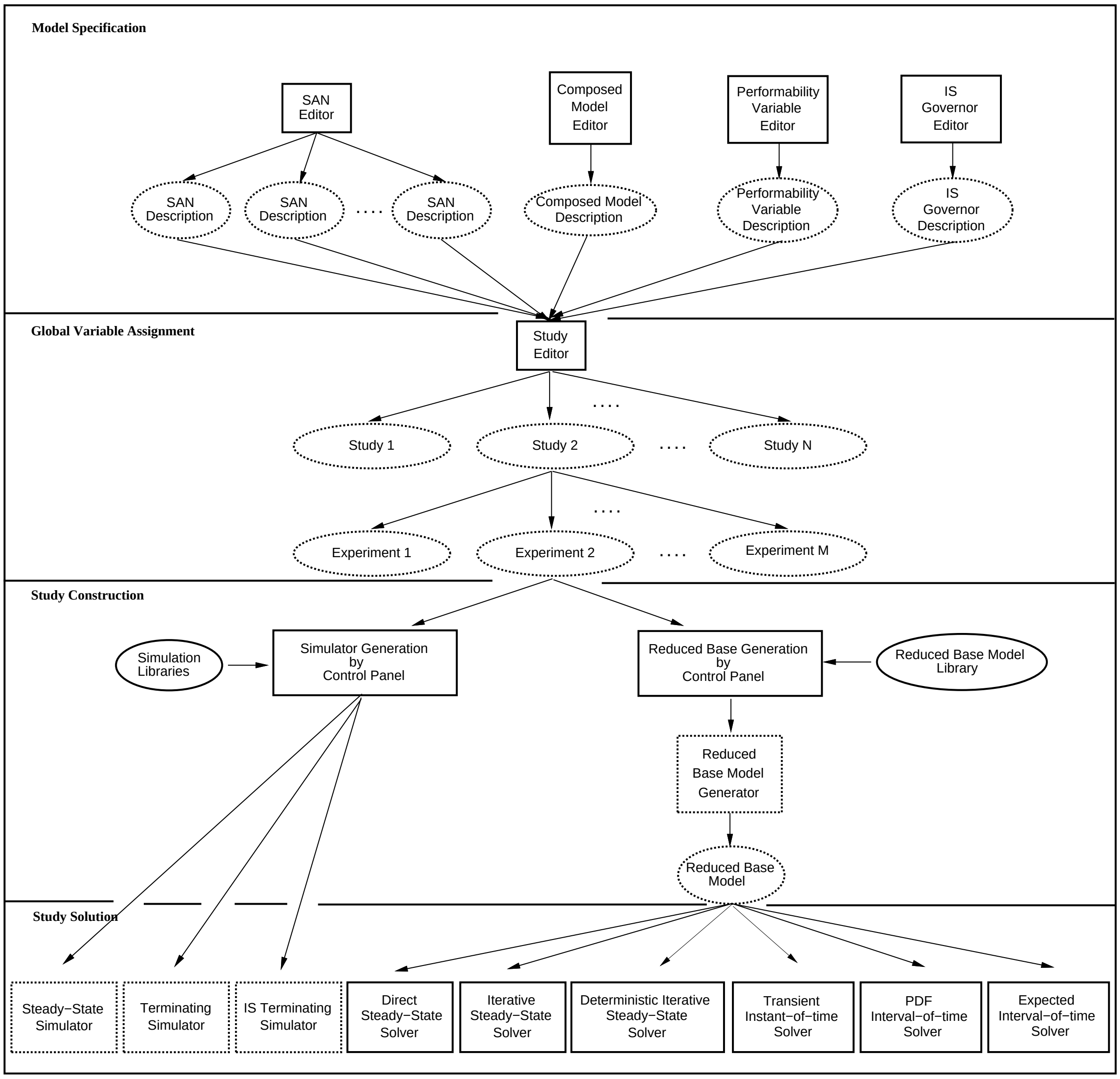

Figure A.1: Organization of *UltraSAN*.

model specifications to be parameterized through the use of global variables in definitions of model components in the SAN editor.

Global variable assignment

In the global variable assignment phase, values are assigned to global variables. A model specification, together with the assignment of a single value to each global variable, is called an experiment. A group of related experiments is called a study. *UltraSAN* provides the study editor to make it easy to specify groups of related experiments.

Study construction

The study construction phase is the process of converting a specified model and a global variable assignment into a format amenable to computer solution via analytic or simulation-based techniques.

Analytical models are constructed by automatically generating a Markov process representation of the model. The size of the generated Markov process is determined in part by the definition of state. In standard stochastic Petri net models, the standard definition of the state space is the set of tangible reachable markings. Realistic models of modern systems usually have very large state spaces. This problem has been widely recognized, and various approaches have been devised to handle the explosive growth [88, 78]. *UltraSAN* uses an innovative largeness-avoidance approach called reduced base model construction [5]. This technique uses information about the performance measures and symmetries in the model to directly generate a stochastic process that is often much smaller than the set of reachable markings and yet is sufficient to support the specified performance measures.

Experience has shown that reduced base model construction is effective for medium to large systems that have replicated subsystems [54].

When the developed model is intended to be solved using simulation, *UltraSAN* does not generate a state-level representation. For this solution method, study construction is accomplished by linking the model description files generated in model specification and global variable assignment to the simulation library. The result is an executable simulation of the model. Essentially, simulation is accomplished by assigning events to activity completions and using the rules of SAN execution to determine the next state. Data structures designed for reduced base model construction [89] make simulation more efficient. Symmetries in the model that allow lumping in state-space construction also help reduce the amount of future events list management necessary for each state change.

Study construction is carried out using the control panel. After verifying that the model description is complete, the control panel builds the reduced base model generator for each experiment to be solved analytically and builds executable simulators for experiments where simulation-based solution is desired. The control panel can then be used to run the reduced base model generator for all experiments that are to be solved analytically. In this case, the control panel can distribute the jobs to run on a user-specified list of other available machines. This "multiple runs" feature makes it easy to construct a study with little effort on the part of the user.

Study solution

The last phase of the modeling process is study solution. If analytic solution is intended, the reduced base model generator was executed during study construction to produce the reduced base model. This representation of the model serves as input to the analytic solvers. Since a single solution method that works well for all models and performance variables is not available, *UltraSAN* utilizes a number of different techniques for solution of the reduced base model. To meet the goal of providing a test-bed for research into new solution techniques, the interface between the reduced base model generator and the analytic solution modules is well defined, making it easy to add new solvers as they become available.

The package currently offers six analytic solution modules. The direct steady-state solver uses LU-decomposition. The iterative steady-state solver uses successive overrelaxation (SOR). The deterministic iterative steady-state solver is used to solve models that have deterministic as well as exponential activities. It uses SOR and uniformization. The transient instant-of-time solver uses uniformization to evaluate performance measures at particular epochs. All of these solvers produce the mean, variance, probability density, and probability distribution function (PDF). The PDF interval-of-time solver uses uniformization to determine the distribution of accumulated reward over a specified interval of time. Similarly, the expected interval-of-time solver uses uniformization to compute the expected value of interval-of-time and time-averaged interval-of-time variables.

Three solution modules are available for simulation. The steady-state simulator uses the method of batch means to estimate the mean and variance of steady-state measures. The terminating simulator uses the method of independent replications to estimate the mean and variance of instant-of-time, interval-of-time, and time-averaged interval-of-time measures. The third simulator is useful when the model contains events that are rare but significant with respect to the chosen performance variables. In this case, traditional simulation will be inefficient due to the rarity with which an event of interest occurs. For problems like this, *UltraSAN* includes an importance sampling simulation module [90] that estimates the mean of a performance measure. The importance sampling simulation module is flexible enough to handle many different heuristics, including balanced failure biasing, and is also applicable to non-Markovian models [91].

The study solution phase is made easier by the control panel multiple runs feature. After study construction, the control panel is used to run the reduced base model generator for all experiments solved analytically. After the state spaces for the experiments have been generated, all experiments can be solved by automatically distributing the solver or simulation jobs to run on all available machines.

These four phases, model specification, global variable assignment, study construction, and study solution are used repeatedly in the modeling process. All phases are supported by a graphical user interface, which facilitates specification of families of parameterized models and solution by many diverse techniques. The following four sections detail the underlying theory and steps taken in each phase of the process.

A.4 Model Specification

The model specification steps are specification of SAN models of the system, composition of the SAN models together to form a composed model, specification of the desired performance, dependability, and performability variables, and, if needed, specification of the "governor" to be used in importance sampling simulation. Completion of these three or four steps results in a model representation that can be used to generate an executable simulation or state-level model representation, once values are assigned to global variables.

Input to each of the four editors is graphical and X-window-based. Detailed instructions concerning the operation of each editor cannot be given due to space limitations, but they can be found in [92]. However, a brief description of what must be specified for each step follows.

SAN model specification

The SAN editor (see Figure A.3) is used to create stochastic activity networks (SANs) that represent the subsystems of the system being studied. Before describing its operation, it is useful to briefly review the primitives that make up a SAN and how these are used to build system and component models. SANs consist of places, activities, and gates. These components are drawn and connected using the SAN editor and defined through auxiliary editors that pop up on the top of the SAN editor.

The functions of the three model primitives are as follows. *Places* are as in Petri nets. Places hold tokens, and the number of tokens in each place is the *marking* of the SAN. The interpretation of the number of tokens in a place is left to the discretion of the modeler. Sometimes tokens are used to represent physical quantities such as the number of customers in a queue or the number of operational components in a system. On the other hand, it is often convenient to use the marking of a place to represent the state of a network or the type of a customer.

Activities are used to model delays in a system. *Timed* activities have a (possibly) marking-dependent *activity time distribution*, a (possibly generally distributed) probability distribution function that describes the nature of the delay represented by the activity. The special situation of a zero delay is represented by the *instantaneous* activity.

Cases are used to model uncertainty about what happens upon completion of an activity. Each activity has one or more cases and a marking-dependent case distribution. A *case distribution* is a discrete probability distribution over the cases of an activity. Using cases, the possible outcomes of the completion of a task can be enumerated, and the probability of each outcome is specified. This modeling feature is useful for representing imperfect coverage or routing probabilities characteristics. They are closest in nature to the "probabilistic arcs" of generalized stochastic Petri nets but differ in that the probabilities assigned to cases can be dependent on the global marking of the SAN.

Gates serve to connect places and activities and are an important part of the modeling convenience offered by SANs. *Input* gates specify when activities are enabled, and what happens to the input places when the activity completes. The predicate of an input gate is a Boolean function of its input places, which when true causes the activity to be enabled. Each input gate also has a function that specifies how the marking of the input places is updated when the activity completes. In most stochastic Petri nets (e.g., GSPNs), enabling conditions and marking updates are specified using networks of immediate transitions and places. Input gates allow functional descriptions of the enabling predicate and marking updates, which is more convenient for specifying complicated enabling conditions and token movements that are common in models of real systems. *Output* gates allow functional specifications of how the marking of output places is updated upon completion of an activity. This flexibility is enhanced when output gates are used in combination with cases. Introduced first in SANs in [93], the utility of gates has become apparent, and similar features have been incorporated into other extensions to stochastic Petri nets, such as the guards in SRNs [8].

Another important feature in the model specification tools is the incorporation of "global variable" support. In the SAN editor, global variables may be used in the definition of initial markings of places, activity time distributions, case distributions, gate predicates, and gate functions. Global variables can be thought of as constants (either integer or floating point) that remain unspecified during model specification and that are varied in different experiments. It is important to point out that the use of global variables is not limited to simple assignments. From its inception, *UltraSAN* has allowed SAN component definitions to make full use of the C programming language. The use of variables global to the model has been built on top of this capability. One ramification of this architecture is that the modeler is free to define SAN components in terms of complicated functions of one or more global variables or change the flow of control in a gate function, rather than simply assigning variables to component parameters.

The fault-tolerant multicomputer system discussed in [53] serves as a good example to illustrate the modeling process in *UltraSAN*. Although this system is a pure dependability model, it incorporates some interesting features that challenge a model description language. As in [53], a multicomputer consists of multiple computer modules connected via a bus, where each module is as shown in Figure A.2. Each computer module consists of three memory modules, three CPUs, two I/O ports, and an error handler. Each computer module is operational if at least two memory modules, two CPUs, one I/O port, and the error handler are operational. The multicomputer is considered to be operational if at least one of the computer modules is operational. As given in [53], reliability is modeled at the chip level, and each chip has a failure rate of 1×10^{-7} failures per hour or 8.766×10^{-4} failures per year.

An interesting feature of the system is that different coverage factors have been assigned to failures at each level of redundancy. These coverage factors are listed in Table A.1, along with the global variables used to represent them. For example, if a RAM chip fails and there are spares available, there is a 0.998 probability that it is successfully replaced by a spare. However, with probability 0.002, this failure causes the memory module to fail. The failure of a memory module is covered (assuming an available spare) with probability 0.95. But with probability 0.05, the memory module failure causes a computer to fail. Finally, if a computer fails, and there is an available spare, the multicomputer fails with probability 0.05. So, in the case where there are available spares at each level of redundancy, the failure of a RAM chip causes the entire multicomputer to fail with probability $0.002 \times 0.05 \times 0.05 = 5 \times 10^{-6}$.

However, if spares are not available at all levels, then the coverage probability is different. If, for example, there are no spare RAM chips, the probability that a RAM chip failure causes the multicomputer to fail is $0.05 \times 0.05 = 0.0025$, since a RAM chip failure after the spares have been used causes the memory module to fail. For this system, the coverage probabilities clearly depend on the state of the system at the time of a component failure. Later, we will show how to model this system feature using marking-dependent case probability distributions.

Figure A.3 shows the SAN model of the memory module in each computer module. The SAN has two activities, called *memory_chip_failure* and *interface_chip_failure*. The input places of *memory_chip_failure* are those connected to input gate *IG1*, namely *computer_failed*, *memory_failed*, and *memory_chips*. As can be seen from Table A.2, *IG1* holds (its predicate is true) if there are at least thirty-nine tokens in *memory_chips*, at most one

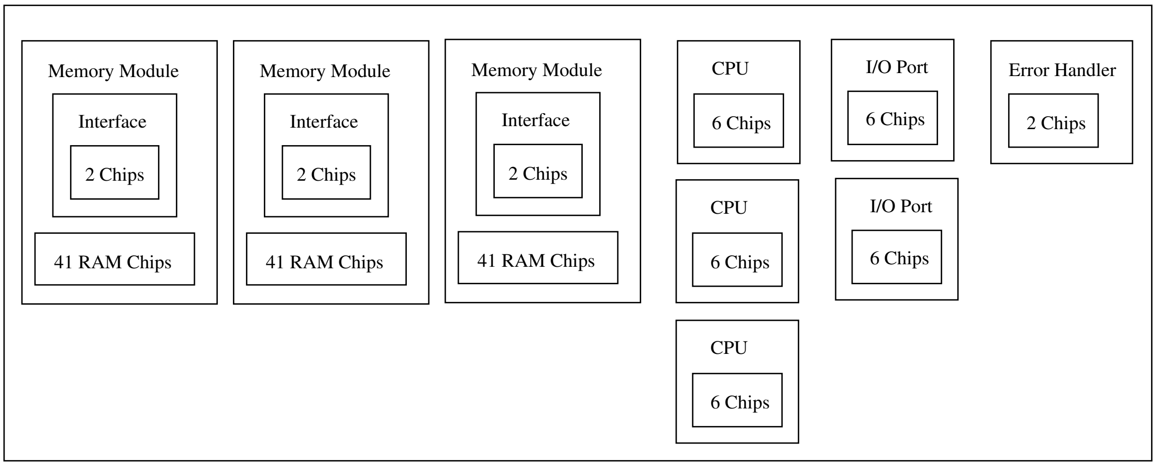

Figure A.2: Block diagram of fault-tolerant computer

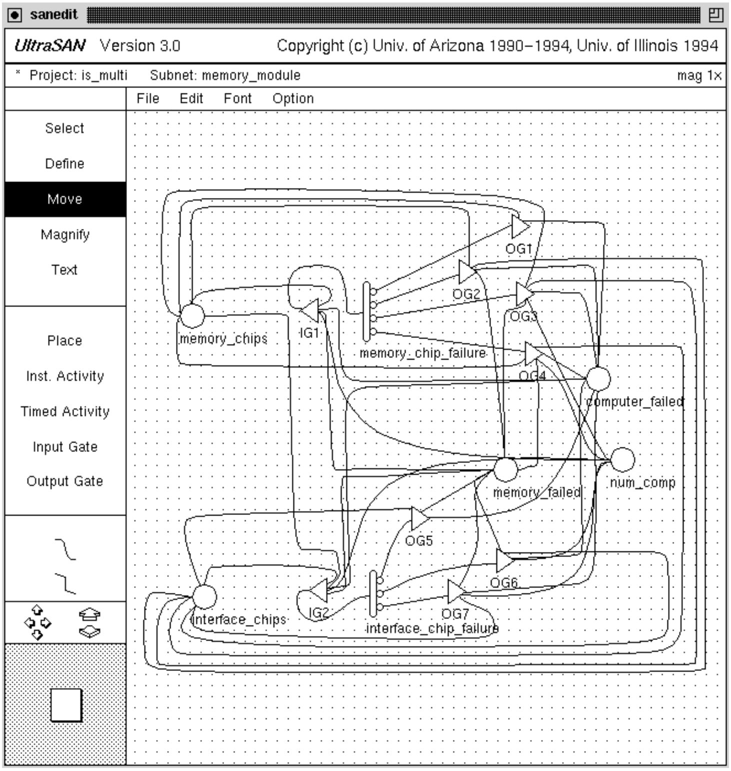

Figure A.3: SAN model of a memory module, as specified in the SAN editor.

token in *computer_failed*, and at most one token in *memory_failed*. Under these conditions, a memory chip failure is possible, and activity *memory_chip_failure* is enabled. The function of input gate *IG1* is the identity function, because the marking of the input places of *memory_chip_failure* depends on the outcome of the failure. To accommodate this condition, the update of the marking is done by the output gates connected to the cases of the activity.

Each of the activities in the memory module SAN has multiple cases, representing the various failure modes of memory chips and interface chips. Connected to each case is an output gate that updates the marking according to the failure mode represented by the case. The output places of activity *memory_chip_failure* are those places connected to the output gates that are connected to the cases of the activity. One can see from Figure A.3 that all of the places in the memory module SAN are output places of *memory_chip_failure*.

Table A.4 shows the parameterized case distribution for *memory_chip_failure*. Global variables have been used so that the impact of perturbations in the coverage factors can be evaluated (see Section A.8). The first case corresponds to a covered RAM chip failure. The second case corresponds to a RAM chip failure that is not covered at the memory module level, but the failure of the memory module is covered at the computer level. In the third case, one of two events occurs as a result of an uncovered RAM chip failure. In the first event, a RAM chip failure that is uncovered at the memory module level propagates to the computer level, causing one computer to fail, but the computer failure is covered at the multicomputer level. In the second event, the memory module failure resulting from the uncovered RAM chip failure would have been covered, but it exhausts the redundancy at the memory module level, and the resulting computer failure is not covered at the

Table A.1: Coverage probabilities in multicomputer system

Component	Coverage Probability	Global Variable Name
RAM Chip	0.998	RAM_cov
Memory Module	0.95	mem_cov
CPU Unit	0.995	CPU_cov
I/O Port	0.99	io_cov
Computer	0.95	comp_cov

Table A.2: Example input gate definitions

Gate	*Definition*
IG1	Predicate *(MARK(memory_chips) > 38) && (MARK(computer_failed) < 2) && (MARK(memory_failed) < 2)*
	Function *identity*
IG2	Predicate *(MARK(interface_chips) > 1) && (MARK(memory_failed) < 2) && (MARK(computer_failed) < 2)*
	Function *MARK(memory_chips) = 0;*

Table A.3: Example output gate definition

Gate	*Definition*
OG2	*MARK(memory_chips) = 0;* *MARK(interface_chips) = 0;* *MARK(memory_failed) ++;* *if (MARK(memory_failed) > 1)* *MARK(computer_failed) ++;*

multicomputer level, leading to system failure. The two events were combined into one case in an effort to reduce the state space. Finally, the last case corresponds to a RAM chip failure that is not covered at any level, causing the entire multicomputer to fail.

The case probabilities are marking-dependent, because the coverage probabilities in the multicomputer system depend on the availability of spare components. To illustrate, consider the second case of *memory_chip_failure*. Using the coverage factors from Table A.1 and assuming a spare memory module, the probability that the event associated with this case occurs is $(1-\ RAM\_cov)\times\ mem\_cov = 0.002 \times 0.95 = 0.0019$ when there is at least one spare RAM chip available, but it increases to $mem\_cov = 0.95$ when there are no spare RAM chips left. There are forty-one RAM chips, two of which are spare. The number of tokens in place *memory_chips* represents the number of working RAM chips. Therefore, a correct model of the coverage probabilities requires a case probability definition such as the one shown in Table A.4. For the second case, if the marking of place *memory_chips* is equal to thirty-nine, then there are no spares available, and the probability of this case is set to 0.95. Otherwise, there are spares available and the probability is set to 0.0019.

Table A.3 shows an example of an output gate description. *OG2* is connected to the second case of activity *memory_chip_failure*, which was described above. The function of *OG2* sets the marking of places *memory_chips* and *interface_chips* to zero, disabling both activities in the memory module SAN. Then the marking of *memory_failed* is incremented, indicating a memory module has failed. If the marking of *memory_failed* is now greater than one, then the computer can no longer operate, since it has only one functioning memory module left, but requires two to be operational. This condition is checked by the *if* statement in the function of *OG2*, and *computer_failed* is incremented if more than one memory module has failed, indicating that the computer has failed.

The activity time distributions, case distributions, and gate predicates and functions are all defined using auxiliary editors that pop up over the SAN editor. The definitions may be entered exactly as they are shown in the tables. The tables of component definitions in this paper have been created by the automatic documentation facility that is available from the control panel of *UltraSAN*. The tables are written in LaTeX and are generated from the model description files.

After SAN models of the subsystems have been specified, the composed model editor is used to create the system model by combining the SANs using replicate and join operations.

Composed Model Specification

In the second step of model specification, the composed model editor is used to create a graphical description of the composed model. A composed model consists of one or more SANs connected to each other through common places via replicate and join operations.

The replicate operation creates a user-specified number of replicas of a SAN. The user also identifies a subset of the places in the SAN as common. The places not chosen to be common will be replicated so they will be distinct places in each replica. The common places are not replicated, but are shared among the replicas. Each replica may read or change the marking of a common place. The join operation connects two or more SANs together through a subset of the places of the SANs that are identified as common.

As in the SAN editor, auxiliary editors pop up over the composed model editor to allow one to specify the details of replicate and join operations. For example, the replicate editor lets one specify the number of replicas and the set of places to be held common among the replicas.

The composed model for the multicomputer system is shown in Figure A.4. As can be seen from the figure, a model of a single computer is composed by joining three replicas of the memory module model with models of the CPU modules, I/O ports, and the errorhandler. The computer model is then replicated to produce a model of the multicomputer system. The place *computer_failed* is an example of a common place. It is held common at the system level, where it can be accessed by all subsystems, so that they may update the number of failed computers in the event of a coverage failure.

Performance, Dependability, and Performability Variable Specification

The third step in model specification is to use the performability variable editor to build reward structures from which the desired performance measures may be derived. In short, to specify a performability variable one defines a reward structure and chooses a collection policy. Variables are typed and often referred to according to the collection policy. The three types of variables supported in *UltraSAN* are "instant-of-time," "interval-of-time," and "time-averaged interval-of-time." *Instant-of-time* variables are obtained by examining the value of the associated reward structure at a particular time t. *Interval-of-time* variables measure the reward accumulated in an interval $[t, t+l]$. *Time-averaged interval-of-time* variables divide the accumulated reward by the length of the interval. For a thorough discussion of the concept and application of these variables, see [18].

For simulation models, "activity variables" can also be defined. *Activity variables* measure the time between completions of activities. Using an activity variable, one can measure

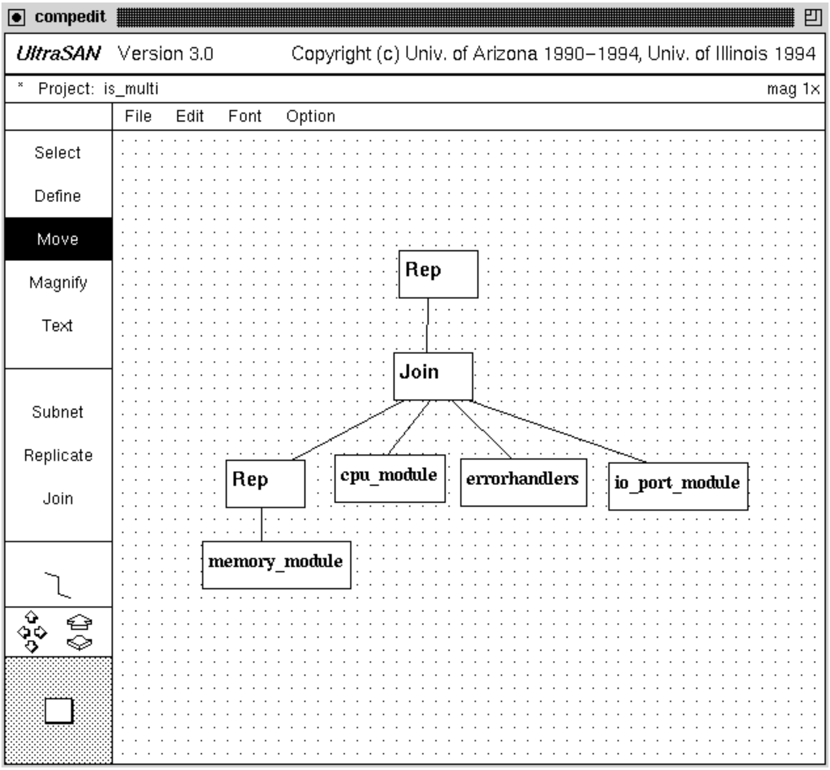

Figure A.4: Composed model for multicomputer, as specified in the composed model editor.

the mean and variance of the time until first completion, the time between the tenth and twentieth completions, etc., using the terminating simulator. The steady-state simulator can be used to measure the mean and variance of the time between consecutive completions as time goes to infinity.

Importance Sampling Governor Specification

For problems where traditional simulation methods are inefficient due to rare but significant events with respect to the chosen performance variables, importance sampling may be used.

Importance sampling is a variance reduction technique where the probability measure on the sample path space is replaced with another probability measure with the intent of improving the efficiency of the simulation. Typically this change of measure is induced by altering the stochastic components of the simulation model. To compensate for taking observations from an altered model, the observations are weighted by the likelihood ratio. This ratio is the ratio of the probability of the sample path under the orginal probability measure to the probability of the same sample path under the new probability measure. This weighting is required to retain an unbiased estimator. For the likelihood ratio to be valid, the new probability measure that is induced by the changes to the stochastic components of the simulation model must assign nonzero probability to all paths that have nonzero probability under the original measure.

In this way, importance sampling can be used to focus the simulation effort on sample paths leading to the event. The stochastic components of the simulation model are altered to increase the likelihood of encountering the rare event. A good example of a problem where importance sampling can help is estimating the unreliability of dependable systems [94, 95, 91]. In this case, the model is altered to accelerate component failures with the intent of increasing the probability of system failure. The cited research shows the utility of varying the acceleration applied to component failures depending on the evolution of a sample path. This approach is one heuristic for evaluating dependable systems. For simulations of other types of systems, different heuristics will be required, since no universally applicable heuristic has been discovered.

In *UltraSAN*, an importance sampling governor [90] is used to specify the heuristic. The governor is similar to a finite state machine, where each state corresponds to a different set of activity definitions. That is to say, in each governor state the user may redefine the activity time and case distribution of each timed activity in each SAN in the composed model. There are two restrictions that must be met for the simulation to be valid. All activity time distribution functions must have closed form complementary distribution functions, and case distributions cannot be altered to give zero probability to cases that have positive probability in the original model. Both of these conditions are checked by the importance sampling simulator at run time.

The governor changes state based on the evolution of the simulation. Each state has a transition function consisting of a sequence of Boolean functions of the markings of the SANs in the composed model (called predicates), paired with a governor state name. When the marking of the model is updated, the transition function for the current governor state is checked. The first predicate that holds causes the governor to change to the associated governor state. When the governor changes state, the activity definitions associated with the new state are applied to the model. Activities whose definitions are changed by the governor transition are reactivated so that the new definition takes effect immediately.

When an activity is reactivated its completion time is sampled from the new activity time distribution function. The details of this process are available in [90].

The utility of the governor is that dynamic importance sampling heuristics may be experimented with, where the bias on the model is altered according to the evolution of a sample path. Additional flexibility is available through the use of marking-dependent governor state definitions.

This approach differs from that taken in other tools, which are specialized to apply a single heuristic. The automatic application of a single heuristic is a feature found in more specialized tools such as SAVE [96], which is designed to handle Markovian dependability models. SAVE applies balanced failure biasing [97], a provably robust technique for importance sampling simulation of models representable in the SAVE model description language. Since *UltraSAN* is intended to handle a broader class of systems, the use of a single heuristic for automatic importance sampling is not practical.

After the model is specified, the next step is to assign values to the variables using the study editor.

A.5 Global Variable Assignment

The *study editor* facilitates assignment of numerical values to the global variables used in a model specification. In the process of assigning values, it provides a convenient way to organize the project in terms of multiple studies and experiments.

Conceptually, a *study* is analogous to a blank folder which can be used to collect experiments. The user can assign any meaning to a study. On the other hand, an experiment is a particular instantiation of the specified model. It consists of the model specification and a set of numerical values assigned to the global variables. Such a set of numerical values is called an *assigned set*.

Studies are created by invoking the study editor following model specification. In the study editor, studies can be added or deleted from the project, and experiments belonging to studies can be generated by assigning values to the global variables through either the range editor or the set editor. The *range editor* is used to generate experiments within a study where each global variable takes on either a fixed value or a sequence of values over a range defined by an interval and a fixed increment. The sequence of numerical values is specified by selecting an initial value, a final value, and an increment factor, which can be additive or multiplicative. The range editor creates assigned sets by evaluating the cartesian product of the values specified for all global variables. Despite its limited scope, the range editor is very efficient for generating experiments for many of studies. When more flexibility is required, the set editor may be used.

The *set editor* is capable of generating experiments for all types of studies. It is capable of assigning any arbitrary sets of numerical values to the global variables. The set editor provides such a capability by allowing the user to directly add, delete or edit an assigned set. Furthermore, the set editor provides facilities for importing (reading) and exporting (writing) formatted files of assigned sets. This enables the user to assign sets to the global variables by editing a file (outside of *UltraSAN*, via "vi" for example) in a fixed format. Alternatively, the import file may be generated by a user's program. Once created, the import file can be read by the set editor, which then generates the corresponding experiments.

Upon reading assigned sets from an import file, the user can add, delete or edit any set. The export facility is useful when a user wants to use the assigned sets of one study in another study. This capability is desirable when the sets of values to be assigned to a study differ with the already assigned sets of another study in only a few elements. The user can first export the assigned sets of the existing study and then after making the changes can import the file for the new study.

After specifying the global variable assignments via the study editor, the next step is study construction.

A.6 Study Construction

Depending on model characteristics, experiments in a study can be solved by analysis or simulation. Specifically, analytic solution can be used when activity time distributions are exponential, activities are reactivated often enough to ensure that their rates depend only on the current state, and the state space is not too large relative to the capacity of the machine. In this case, the underlying stochastic process for the model is a Markov process. In addition, mixes of exponential and deterministic activities can be handled, as long as no more than one concurrently enabled deterministic activity is enabled in any reachable stable marking. Otherwise, simulation must be used.

If analysis is the intended solution method, a stochastic process is constructed that supports solution of the variables in question. In most stochastic Petri net based tools, the set of reachable stable (also known as "tangible") markings is made to be the set of states in the Markov process. This choice causes two problems: 1) the state space generated for realistic systems is often unmanageably large, and 2) not all variables can be written in terms of this process. In regard to the second problem, impulse rewards may be specified in a way that requires distinguishing between two different activity completions that result in the same transition in a Markov process constructed in this way.

One potential solution to this problem is to construct a process which keeps track (in its notion of state) of most recently completed activities, as well as reached stable markings. This process (called the activity-marking behavior, in [5]), supports all reward variables, but will result in state spaces that are very large. Alternatively, one can construct a stochastic process that is tailored to the variable in question, and exploits symmetries inherent in the model. When such symmetries occur, this approach, known as "reduced base model construction" can significantly reduce the size of the process that must be solved to obtain a solution. Symmetries are identified through the use of the replicate and join operation in the composed model editor (as described in the model specification section). This resulting state-level representation is known as a *reduced base model* [5].

The reduced base model is constructed from the model specification, which is known formally as a "composed SAN-based reward model." In order to understand the (non-standard) notion of state used in reduced base model construction, a more formal description of the model specification must be given. In particular, during model construction, SAN-based reward models are constructed by application of the replicate or join operations on input "SAN-based reward models" (SBRMs). Formally, a *SAN-based reward model* consists of a SAN, a reward structure defined on the SAN, and a set of common places (a subset of the set of places in the SAN). Reward structures are defined at the SAN level, rather than composed model, to insure that replicated SANs have the same reward structure, and hence behave symmetrically with respect to the specified performance variables.

The resulting SBRM used in model construction is referred to as a *composed SAN-based reward model*, since it is the result of the composition of multiple SBRMs.

As alluded to in the model specification section, the replicate operation is useful when a system has two or more identical subsystems. The effect of the operation depends on a chosen set of places which is a subset of the common places of the input SBRM. These places are not replicated when the operation is carried out (informally, they are "held common" in the operation), and hence allow communication between the replicated submodels. The resulting SBRM consists of a new SAN, reward structure, and set of common places. The SAN is constructed by replicating the input SAN some number of times, holding the chosen set of places common to all SANs. The reward structure is constructed by assigning: 1) an impulse reward to each replicate activity equal to its value in the original model and, 2) a reward rate to each marking in the new SBRM equal to the sum of the rates assigned to the marking of each replica of the input SBRM. Finally, the set of common places in the new SBRM is taken to be the set of places held common during the operation.

Composition of multiple distinct SBRMs is done using the join operation. The *join* operation acts on SBRMs and produces a SBRM. The new SBRM is produced by identifying certain places in the different input SANs as common in the new SAN. These common places allow arbitrary communication between the SANs that are joined; the only restriction being that the initial marking of the places to be combined is identical. The reward structure in the new SBRM is constructed by assigning: 1) an impulse reward to each activity in the joined SBRM equal to the reward of the corresponding activity in the input SBRM, and 2) a rate reward to each marking in the joined SBRM equal to the sum of the rate rewards of the markings in the input SBRM whose union constitutes the marking of the joined SBRM.

The set of common places of the new SBRM is the subset of the set of common places of each input SBRM that is made common, with zero or more models that are joined together.

The composed SBRM constructed in this way is input to the reduced base model construction procedure to generate the reduced base model. The notion of state employed is variable and depends on the structure of the composed model. One can think of the state as a set of impulse and rate rewards plus a state tree [89], where each node on the state tree corresponds in type and level with a node on the composed model tree (as specified in the composed model editor). Each node in a state tree has associated with it a subset of common places of the corresponding node in the composed model diagram. These places are those that are common at the node but not at its parent node. The saving in state-space size, relative to the standard state generation approach, is due to the fact that at each replicate node in the tree, we keep track of the *number* of submodels in particular markings, rather than the marking of *each* submodel. Proof that this notion of state produces Markov processes whenever the standard state generation algorithm can be found in [5].

Figure A.5 shows the state tree of a particular state of the reduced base model for the multicomputer example. Places *interface_chips* and *memory_chips* are not common at the replicate node *R2*. Therefore, as shown in Figure A.5, they are unique for each memory module and can be different for different replica copies. At the join node, the set of common places is the union of the common places associated with the SANs of cpu_module, errorhandlers, io_port_module, and replicate node R2. Finally, among these places, *computer_failed* is common to all computer modules. The other places: *memory_failed*, *cpus*, *errorhandlers*, and *ioports* are not common and, hence, are unique to each computer module.

Table A.4: Activity case probabilities for SAN model *memory_module*

Activity	*Case*	*Probability*
interface_chip_failure	1	*0.95*
	2	*0.0475*
	3	*0.0025*
memory_chip_failure	1	*if (MARK(memory_chips) == 39)* *return(0.0);* *else* *return(0.998);*
	2	*if (MARK(memory_chips) == 39)* *return(0.95);* *else* *return(0.0019);*
	3	*if (MARK(memory_chips) == 39)* *return(0.0475);* *else* *return(0.000095);*
	4	*if(MARK(memory_chips) == 39)* *return(0.0025);* *else* *return(0.000005);*

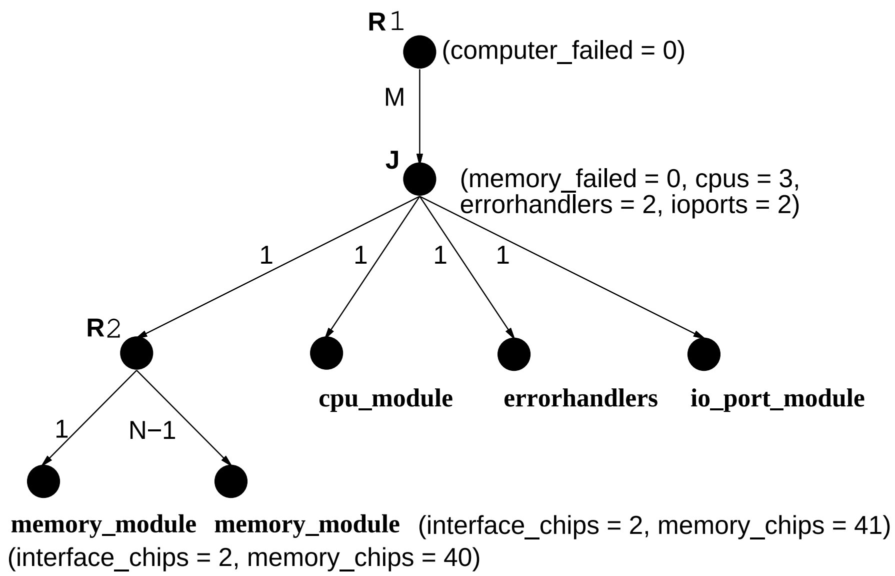

Figure A.5: The state tree representation

A reduced base model employing this notion of state can be generated directly without first generating the detailed (marking or activity-marking) behavior. This is done by first creating the state tree corresponding to the initial marking of the SAN and then generating all possible next states by executing each activity that may complete in the state. A new state tree and impulse reward are generated corresponding to each possible next state that may be reached. For each possible next state, a non-zero rate (or probability, if a deterministic activity is enabled in the state, see [98]) from the original state to the reached state is added to the set of rates to other states from the originating state. If the reached state is new, then it is added to the set of states which need to be explored. Generation of the reduced base model proceeds by selecting states from the set of unexplored states and repeating the above operations. The procedure terminates when there are no more states to explore. The resulting reduced base model serves as input to the analytic solvers, which determine the probabilistic nature of the selected performability variables.

Unlike the analytic solvers, which require a reduced base model, the simulators execute directly on the assigned model specifications. As depicted in Figure A.1, three types of simulation programs can be generated: a *steady-state simulator* to solve long-run measures of performability variables, a *terminating simulator* to solve instant-of-time or interval-of-time performability variables, and an importance sampling terminating simulator. When simulation is the intended solution method, in the study construction phase the control panel links experiments (model descriptions) with the appropriate simulation library to generate the selected executable simulation programs. In the case of importance sampling simulation, the control panel links the governor description created by the importance sampling editor together with the experiment and importance sampling simulation library.

The next section details how the analytic and simulation solvers are used to determine the probabilistic behavior of the desired performability variables.

A.7 Study Solution

Solution of a set of base models, corresponding to one or more experiments within a study, is the final step in obtaining values for a set of performability variables specified for a model. The solution can be either by analytic (numerical) means or simulation, depending on model characteristics and choices made in the construction phase of model processing. This section details the solution techniques employed in *UltraSAN* to obtain analytic and simulative solutions.

A.7.1 Analytic Solution Techniques

Analytic solvers can provide steady-state as well as transient solutions for the reduced base model generated during study construction. Figure A.1 illustrates the six solvers available for analytic solutions. Three of the solvers, specifically, the *direct steady-state solver*, *iterative steady-state solver*, and *deterministic iterative steady-state solver*, provide steady-state solutions for the instant-of-time variables. The other three solvers, the *transient instant-of-time solver*, *PDF interval-of-time solver*, and *expected interval-of-time solver*, provide solutions for transient instant-of-time and interval-of-time variables. Except for the two interval-of-time solvers, each solver calculates the mean, variance, probability density function, and probability distribution function of the variable(s) for which it is intended to solve. The PDF interval-of-time solver calculates the probability distribution function of reward accumulated during a fixed interval of time $[0, t]$. Likewise, the expected interval-of-time solver calculates the expected reward accumulated during a fixed interval of time. All solvers use a sparse matrix representation appropriate for the solution method employed.

The direct steady-state solver uses LU decomposition [99] to obtain the steady-state solution for instant-of-time variables. As with all steady-state solvers, it is applicable when the generated reduced base model is Markovian and contains a single, irreducible, class of states. LU decomposition factors the transition matrix Q into the product of a lower triangular matrix L and an upper triangular matrix U such that $Q = LU$. Crout's algorithm [100] is incorporated to compute L and U in a memory-efficient way. Pivoting in the solver is performed by using the improved generalized Markowitz strategy, as discussed by Osterby and Zlatev [101]. This strategy performs pivoting on the element for which the minimum fill-in is expected, subject to constraints on the magnitude of the pivot. Furthermore, the matrix elements that are less than some value (called the drop tolerance) are made zero (i.e., *dropped*) to retain the sparsity of the matrix during decomposition. In the end, iterative refinement is used to correct the final solution to a user-specified level of accuracy.

The iterative steady-state solver uses successive-overrelaxation to obtain steady-state instant-of-time solutions. Like the direct steady-state solver, it acts on the Markov process representation of a reduced base model. However, being an iterative method, it avoids fill-in problems inherent to the direct steady-state solver and therefore requires less memory. However, unlike LU decomposition, SOR is not guaranteed to converge to the correct solution for all models. In practice however, this does not seem to be a problem. The implementation solves the system of equations directly, without first adding the constraint that the probabilities sum to one, and normalizes only at the end and as necessary to keep the solution vector within bounds, as suggested in [102]. Selection of the acceleration factor is left to the user and defaults to a value of 1 (resulting in a Gauss-Seidel iteration).

The method used in the deterministic iterative steady-state solver was first developed for deterministic stochastic Petri nets [103, 104] with one concurrently-enabled deterministic transition, and later adapted to SANs in which there is no more than one concurrently-enabled deterministic activity [98]. In this solution method, a Markov chain is generated for marking in which a deterministic activity is enabled, representing the states reachable from the current state due to completions of exponential activities, until the deterministic activity either completes or is aborted. The solver makes use of successive-overrelaxation and uniformization to obtain the solution.

The transient instant-of-time solver and expected interval-of-time solver both use uniformization to obtain solutions. Uniformization works well when the time point of interest is not too large, relative to the highest rate activity in the model. The required Poisson probabilities are calculated using the method by Fox and Glynn [105] to avoid numerical difficulties.

The PDF interval-of-time solver [51] also uses uniformization to calculate the probability distribution function of the total reward accumulated during a fixed interval $[0, t]$. This solver is unique among the existing interval-of-time solvers in that it can solve for reward variables containing both impulse and rate rewards. The solver calculates the PDF of reward accumulated during an interval by conditioning on possible numbers of transitions that may occur in an interval, and possible sequences of state transitions (paths), given a certain number of transitions have occurred. Since the number of possible paths may be very large, an implementation that considers all possible paths would require a very large amount of memory. This problem is avoided in *UltraSAN* by calculating a bound on the error induced in the desired measure by not considering each path, and discarding those with acceptably small (user specified) errors. In many cases as illustrated in [51], selective discarding of paths can dramatically reduce the space required for a solution while maintaining acceptable accuracy. Calculation of the PDF of reward accumulated, given that a particular path was taken, can also pose numerical difficulties. In this case, the solver makes selective use of multiprecision math to calculate several summations. Experience with the implementation has shown that this selective use of multiprecision operations avoids numerical difficulties, while maintaining acceptably fast execution.

A.7.2 Simulative Solutions

As shown in Figure A.1, three simulation-based solvers are available in *UltraSAN*. All three simulators exploit the hierarchical structure and symmetries introduced by the replicate operation in a composed SAN-based reward model to reduce the cost of future event list management. More specifically, multiple future events lists are employed, one corresponding to each leaf on the state tree or, in other words one corresponding to each set of replica submodels in a particular marking. These multiple future event lists allow the simulators to operate on "compound events," corresponding to a set of enabled replica activities, rather than on individual activities. By operating on compound events, rather than individual activities, the number of checks that must be made upon each activity completion is reduced. The details and a precise algorithm can be found in [89]. This algorithm is the basis of the state-change mechanism for all three simulators.

The steady-state simulator uses an iterative batch means method to estimate the means and/or variances of steady-state instant-of-time variables. The user selects a confidence level and a maximum desired half-width for the confidence interval corresponding to each variable, and the simulator executes batches until the confidence intervals for every performability variable satisfy the user's criteria. The terminating simulation solver estimates the means and/or variances for performability variables at a specific instant of time or over a given interval of time. It uses an iterative replication method to generate confidence intervals that satisfy the criteria specified by the user.

The third simulation-based solver is a terminating simulator that is designed to use importance sampling to increase the efficiency of simulation-based evaluation of rare event probabilities. The importance sampling terminating simulator estimates the mean of transient variables that are affected by rare events by simulating the model under the influence of the governor (described in Section A.4), so that the interesting event is no longer rare. The simulator then compensates for the governor bias by weighting the observations by the likelihood ratio. The likelihood ratio is calculated by evaluating the ratio of the likelihood that a particular activity completes in a marking, and that the completion of this activity results in a particular next marking, in the original model to the likelihood of the same event in the governed model. The likelihood calculation is based on the work in [106], adapted for SAN simulation and implemented in *UltraSAN* [90]. It works for models where all activities have closed-form activity time distribution functions. The probability distribution function is needed to handle the reactivations of activities when their definitions are changed by the governor.

A.8 Example Results

The reliability of the multicomputer system example discussed earlier has been evaluated in [53] using a hierarchical evaluation technique based on numerical integration, and using SANs and the *UltraSAN* transient instant-of-time solver in [54]. Both studies focus on long mission times. In this section we discuss the evaluation of the unreliability of the multicomputer for shorter mission times, using the transient instant-of-time solver and the importance sampling terminating simulator.

Since there is no repair capability in the model of the multicomputer system, the system failed state is an absorbing state. Therefore, an interval-of-time variable such as unreliability can be evaluated as an instant-of-time variable. The reward structure for unreliability is simple, consisting of a rate reward of one assigned to the system failed state, with all other states having rate reward zero. Also, all impulse rewards are zero.

To evaluate the unreliability of the multicomputer for a given mission time using the transient instant-of-time solver, one simply generates the reduced base model and runs the solver. The transient instant-of-time solver is designed to handle multiple time points, so it is easy to evaluate the unreliability at a sequence of possible mission times. The answers for earlier instants of time are obtained with little additional effort compared to obtaining the answer for the last time point.

If importance sampling is to be used to evaluate the unreliability, a governor must be constructed to guide the importance sampling simulation. In this case, we chose a heuristic called "approximate forcing with balanced failure biasing [95]." Approximate forcing is based on the forcing technique developed in [94] to ensure that a failure event occurs before the mission is completed. In approximate forcing, failure events are accelerated so that a component failure occurs before the end of a mission with probability 0.8. When combined with balanced failure biasing, not only is the total probability of a failure event before mission completion increased, but the conditional probabilities of particular failure events, given a failure event occurs, are made equal.

A governor that implements approximate forcing with balanced failure biasing consists of two states. In the first state, all failure events are given equal probability of occurring, and the total probability of a component failure before the mission time is increased to 0.8. Since the example model is Markovian, the new activity parameters are not difficult to calculate. The total rate of all failure events must be $\rho = -\log 0.2/t$, where t is the mission time in years, since that was the chosen time scale for this model. To assign equal probability to every possible failure event means equal probability must be assigned to every case of every failure activity.

Therefore, in the first governor state, the case distribution for each activity corresponding to a component failure event is the discrete uniform distribution. The rate parameter for a given failure activity is calculated using the formula $\lambda = c\rho/C$, where c is the number of cases on the activity, and C is the total number of cases in the model. For the multicomputer model with two computers, $C = 58$. For a mission time of thirty days, an activity with four cases would be assigned a rate $(4 \times -\log 0.2 \times 365.25)/(58 \times 30)$. Note that by balancing the probabilities of all possible failure events, we are increasing the conditional probability of a system failure due to an uncovered component failure, relative to the probability of system failure through exhaustion of available spares.

The second governor state is unbiased. Since there is no repair in the model, there are no activities competing with the failure activities, so it is sufficient to allow the model to evolve naturally from this point forward. An overly aggressive biasing scheme at this point would cause the simulation to waste time exploring very unlikely paths to system failure.

The multicomputer model with two computer modules was evaluated for missions from one to 390 days in length, using the transient instant-of-time solver and the importance sampling terminating simulator with the governor described above. The accuracy requested in the solver was 1×10^{-9}, while the simulator was set to run until the estimated half width of a 99% confidence interval was within 10% of the point estimate. The results from the two different solution approaches are plotted in Figure A.6. As can be seen from the figure, the results are in close agreement. There is one errant point from the simulator, but the result obtained from the transient instant-of-time solver is still within the confidence interval.

Using importance sampling, with the same governor as before, the impact of increasing the redundancy at the computer level was evaluated for a mission time of thirty days. As shown in Figure A.7, for a thirty day mission, increasing the redundancy led to an increase in unreliability. Due to the short mission time, coverage failures dominate, so increasing the number of components will make the system less reliable. This is because it is now more likely that some component will fail before the mission ends, which in turn increases the probability of a coverage failure.

The impact of perturbations in the coverage factors for RAM chip failures and computer module failures was evaluated using the transient instant-of-time solver for the multicomputer with two computer modules. These studies were carried out using the study

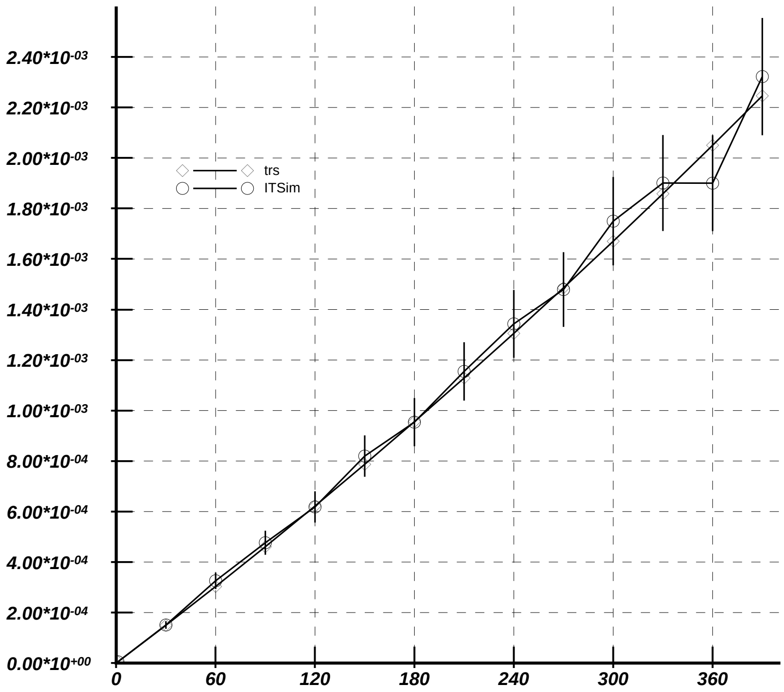

Figure A.6: Unreliability versus mission time (in days).

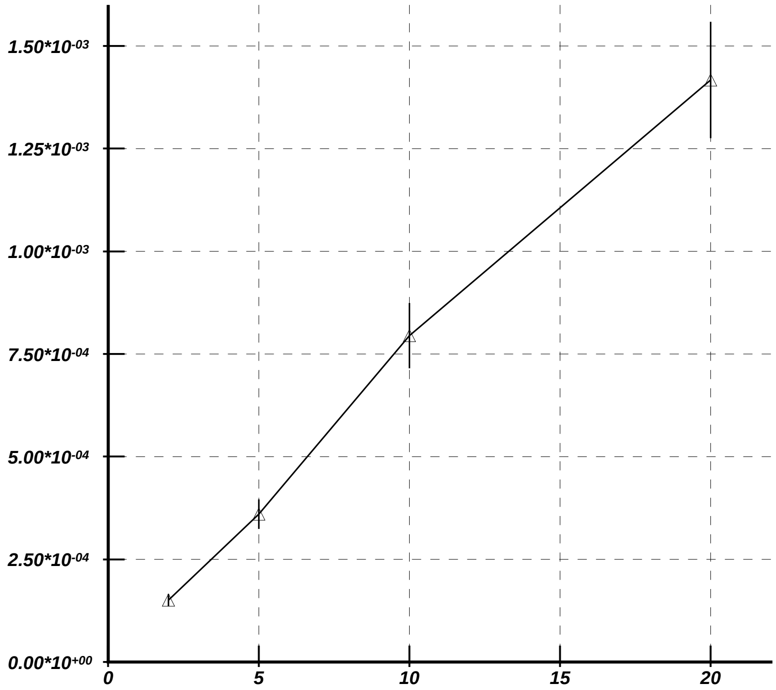

Figure A.7: Unreliability versus number of computer modules.

definitions shown in Table A.5. As can be seen from Figure A.8, the unreliability is not very sensitive to variations in the RAM chip failure coverage. However, variations in the coverage of computer module failures had a large impact on the unreliability, as shown in Figure A.9.

The presented results indicate the importance of accurate estimates of coverage at the computer level, and constitute a strong argument for improvement of this coverage factor. On the other hand, perhaps money could be saved by using a less expensive fault tolerance mechanism to cover RAM chip failures, since the system unreliability seems relatively insensitive to this factor when mission times are short.

A.9 Conclusions

We have described *UltraSAN*, an environment designed to facilitate the specification, construction, and solution of performability models represented as composed stochastic activity networks. *UltraSAN* has been in use for a period of about four years at several industrial and academic sites. Feedback from these users, and the development of new solution methods, has motivated continuous improvement of the software. As it stands today, *UltraSAN* is an environment that incorporates features and capabilities that do much to automate and simplify the task of model-based evaluation, and as evidenced by its application (e.g., [107, 108, 109, 54]), is capable of evaluating the performance, dependability, and performability of "real-world" systems. In addition to this capability, the tool serves as a test-bed in the development of new model construction and solution algorithms.

In particular, the control panel provides a simple interface to the package that enhances productivity and provides useful utilities such as automatic documentation. Models are

Table A.5: Study Editor range definitions for project *multi_proc*

Study	Variable	Type	Range	Increment	Add./Mult.
vary_RAM_coverage					
	CPU_cov	double	0.995	–	–
	RAM_cov	double	[0.75, 1.0]	0.05	Add.
	comp_cov	double	0.95	–	–
	io_cov	double	0.99	–	–
	mem_cov	double	0.95	–	–
vary_comp_cov					
	CPU_cov	double	0.995	–	–
	RAM_cov	double	0.998	–	–
	comp_cov	double	[0.75, 1.0]	0.05	Add.
	io_cov	double	0.99	–	–
	mem_cov	double	0.95	–	–

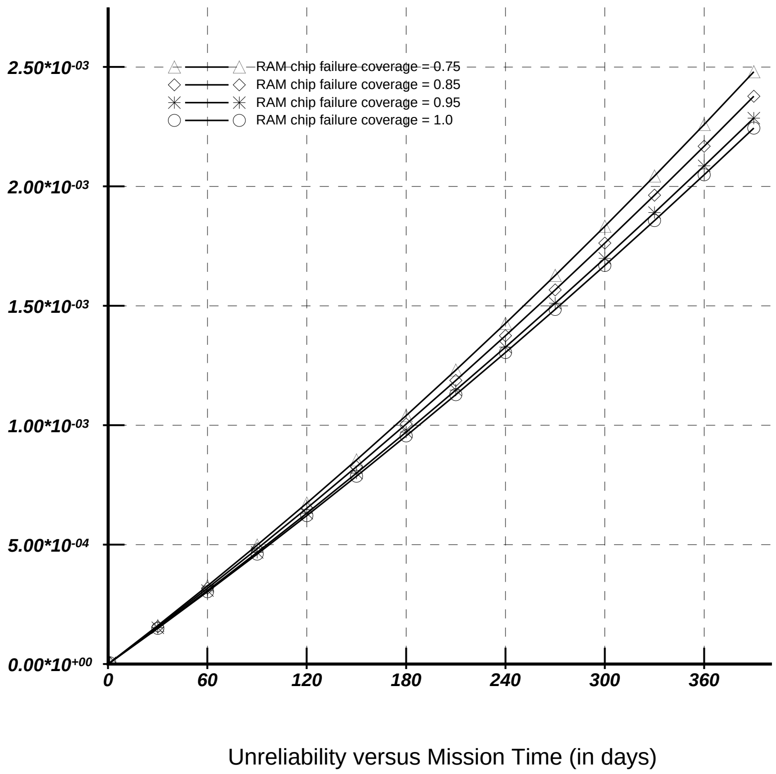

Figure A.8: Impact of perturbation in RAM chip failure coverage factor.

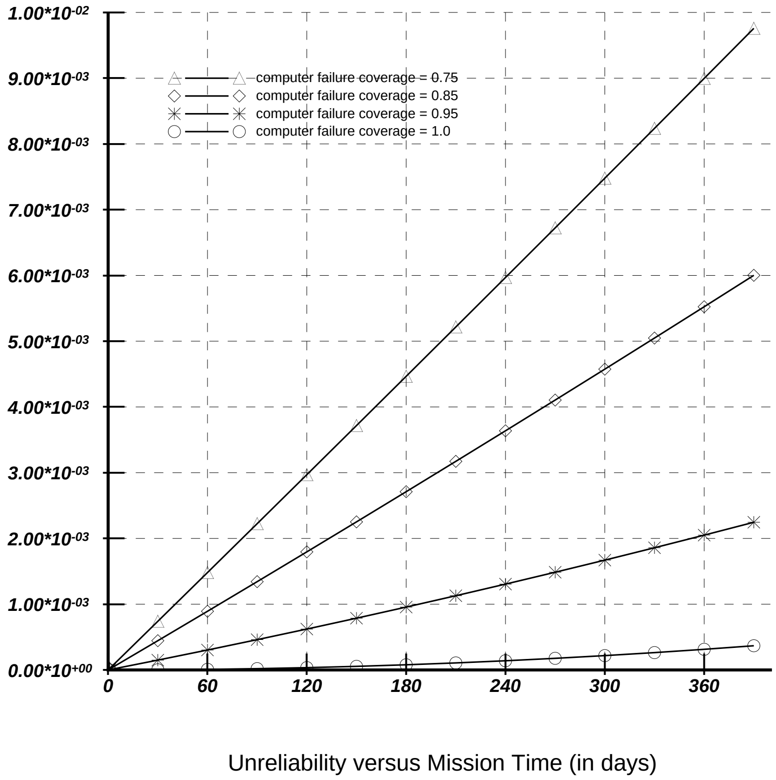

Figure A.9: Impact of perturbation in computer failure coverage factor.

specified as SANs, a model description language capable of describing a very large class of systems. Modeling conveniences such as cases and gates make it easier to specify the complicated behavior common in realistic models of modern computer systems and networks. The generation and solution of large numbers of production runs for a model is made easier through support for model specifications parameterized by global variables. Moreover, hierarchical composed models, created using the replicate and join operations, allow the modeling problem to be broken down into smaller, more manageable pieces. Wherever system symmetries exist, as identified by the replicate operation, they can be exploited during model construction to produce a smaller base model than possible with traditional techniques.

In the area of model construction, reduced base model construction allows *UltraSAN* to handle the typically large state spaces of models of "real-world" systems. The set of systems that can be evaluated analytically using *UltraSAN* has been enlarged by the recent addition of the capability to generate reduced base models for composed SANs that contain a mix of exponential and deterministic timed activities. Construction of models parameterized by global variables is automated through the control panel, which is capable of distributing model construction jobs across a network to utilize other available computing resources.

Both analytic and simulation based solution methods have been implemented in *UltraSAN*. For analytic solution, six different solvers are available to compute performance, dependability and performability measures at an instant of time or over an interval of time, for models with mixes of exponential and deterministic timed activities, and for transient or steady-state. If simulation is to be used, three simulation-based solvers are available. The first two are traditional simulators; a terminating simulator for transient measures and an iterative batching simulator for steady-state measures. The third simulator is an importance sampling terminating simulator that implements importance sampling heuristics represented as governors. Using the control panel, model solution jobs can also be distributed across a network of workstations and carried out in parallel, improving productivity. The fact that three of the six solvers, and the importance sampling terminating simulator, were added after the initial release of *UltraSAN* demonstrates the success of the environment as a test-bed for the development and implementation of new solution techniques.

Acknowledgments

The success of this project is the result of the work of many people, in addition to the authors of this paper. In particular we would like to thank Burak Bulasaygun, Bruce McLeod, Aad van Moorsel, Bhavan Shah, and Adrian Suherman, for their contributions to the tool itself, and the members of the Performability Modeling Research Lab who tested the pre-release versions of the software. We would also like to thank the users of the released versions who made suggestions that helped improve the software. In particular, we would like to thank Steve West of IBM Corporation, Dave Krueger, Renee Langefels, and Tom Mihm of Motorola Satellite Communications, Kin Chan, John Chang, Ron Leighton and Kishore Patnam of US West Advanced Technologies, Lorenz Lercher of the Technical University of Vienna, and John Meyer, of the University of Michigan in this regard. We would also like to thank Sue Windsor for her careful reading of the manuscript.

REFERENCES

[1] A. Movaghar, *Performability modeling with stochastic activity networks*, Ph.D Dissertation, University of Michigan, Ann Arbor, 1985.

[2] J. F. Meyer and A. Movaghar and W. H. Sanders, "Stochastic activity networks: Structure, behavior, and applications," *Proceedings of International Workshop on Timed Petri Nets*, (Torino, Italy), pp. 106–115, July 1985.

[3] M. Molloy, *On the integration of delay and throughput measure in distribution processing model*, Ph.D Dissertation, University of California, Los Angeles, 1985.

[4] S. Natkin, *Reseaux de Petri stochastiques.* CNAN, Paris, 1980.

[5] W. H. Sanders and J. F. Meyer, "Reduced base model construction methods for stochastic activity networks," *IEEE Journal on Selected Areas in Communications*, vol. 9, pp. 25–36, January 1991.

[6] R. A. Howard, *Dynamic Probabilistic Systems, Vol II: Semi-Markov and Decision Processes.* John Wiley & Sons, 1971.

[7] G. Ciardo, J. Muppala, and K. S. Trivedi, "On the solution of GSPN reward models," *Performance Evaluation*, vol. 12, pp. 237–253, 1991.

[8] G. Ciardo, A. Blackmore, P. F. 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*, Springer Verlag, 1993.

[9] M. A. Marsan, G. Balbo, and G. Conte, "A class of generalized Petri nets for performance and evaluation of multiprocessor systems," *ACM Transactions on Computer Systems*, vol. 2, pp. 93–122, May 1984.

[10] V. O. K. Li and J. A. Silvester, "Performance analysis of networks with unreliable components," *IEEE Transactions of Communications*, vol. 32, no. 10, pp. 1105–1110, 1984.

[11] C. L. Yang and P. Kubat, "Efficient computation of most probable states for communication networks with multimode components," *IEEE Transactions of Communications*, vol. 37, no. 5, pp. 535–538, 1989.

[12] E. de Souza e Silva and P. M. Ochoa, "State space exploration in Markov models," in *Proceedings of Sigmetrics/Performance-92*, (Newport, Rhode Island), pp. 152–166, June 1992.

[13] W. K. Grassman, "Finding transient solutions in Markovian event systems through randomization," in *Proceedings of the 1st International Workshop on the Numerical Solution of Markov Chains*, (Newport, Rhode Island), pp. 152–166, June 1990.

[14] G. Florin, C. Fraize, and S. Natkin, "Searching best paths to worst states," in *Proceedings of the 4th International Workshop on Petri Nets and Performance Models*, (Melbourne, Australia), pp. 204–209, Dec 1991.

[15] J. L. Bon and J. Collet, "An algorithm in order to implement reliability exponential approximations," in *Proceedings of the 2nd International Workshop on Performability Modelling of Computer and Communication Systems*, (Le Mont Saint-Michel, France), June 1993.

[16] D. M. Nicol and D. L. Palumbo, "Reliability analysis of complex models using sure bounds," *IEEE Transactions on Reliability*, vol. 44, no. 1, pp. 46–53, 1995.

[17] G. Ciardo, R. German, and C. Lindemann, "A characterization of the stochastic process underlying a stochastic Petri net," *IEEE Transactions on Software Engineering*, vol. 20, pp. 506–515, July 1994.

[18] W. H. Sanders and J. F. Meyer, "A unified approach for specifying measures of performance, dependability, and performability," in *Dependable Computing for Critical Applications* (A. Avizienis and J. C. Laprie, eds.), vol. 4 of *Dependable Computing and Fault-Tolerant Systems*, pp. 215–238, Springer Verlag, 1991.

[19] L. Kleinrock, *Queueing Systems, Vol I: Theory.* John Wiley & Sons, 1975.

[20] C. Moler and C. V. Loan, "Nineteen dubious ways to compute the exponential of a matrix," *Society of Industrial and Applied Mathematics, Review*, vol. 20, pp. 801–837, October 1978.

[21] A. Reibman and K. Trivedi, "Numerical transient analysis of Markov models," *Computers and Operations Research*, vol. 15, no. 1, pp. 19–36, 1988.

[22] A. Reibman, R. Smith, and K. Trivedi, "Markov and Markov reward model transient analysis: An overview of numerical approaches," *European Journal of Operations Research*, vol. 40, pp. 257–267, 1989.

[23] W. J. Stewart, *Introduction to the Numerical Solution of Markov Chains.* Princeton University Press, 1994.

[24] A. Jensen, "Markov chains as an aid in the study of Markov processes," *Skand. Aktuarietidskrift*, vol. 36, pp. 87–91, 1953.

[25] W. K. Grassmann, "Transient solutions in Markovian queues," *European Journal of Operations Research*, vol. 1, pp. 396–402, 1977.

[26] D. Gross and D. R. Miller, "The randomization technique as a modeling tool and solution procedure for transient Markov processes," *Operations Research*, vol. 32, pp. 343–361, 1984.

[27] J. F. Meyer, "On evaluating the performability of degradable computer systems," *IEEE Transactions on Computers*, vol. 22, pp. 720–731, 1980.

[28] J. F. Meyer, "Closed-form solutions of performability," *IEEE Transactions on Computers*, vol. 31, no. 7, pp. 648–657, 1982.

[29] D. G. Furchgott and J. F. Meyer, "A performability solution method for degradable non-repairable systems," *IEEE Transactions on Computers*, vol. 33, no. 6, pp. 550–554, 1984.

[30] L. Donatiello and B. R. Iyer, "Analysis of a composite performance reliability measure for fault-tolerant systems," *Journal of the ACM*, vol. 34, no. 1, pp. 179–199, 1987.

[31] A. Goyal and A. N. Tantawi, "Evaluation of performability for degradable computer systems," *IEEE Transactions on Computers*, vol. 36, no. 6, pp. 738–744, 1987.

[32] B. Ciciani and V. Grassi, "Performability evaluation of fault-tolerant satellite systems," *IEEE Transactions on Communications*, vol. 35, no. 4, pp. 403–409, 1987.

[33] V. G. Kulkarni, V. F. Nicola, R. M. Smith, and K. S. Trivedi, "Numerical evaluation of performability and job completion time in repairable fault-tolerant systems," in *Proceedings of the 16th Int. Symposium on Fault-Tolerant Computing*, (Vienna, Austria), pp. 252–257, July 1985.

[34] R. M. Smith, K. S. Trivedi, and A. V. Ramesh, "Performability analysis: Measures, an algorithm, and a case study," *IEEE Transactions on Computers*, vol. 37, no. 2, pp. 406–417, 1987.

[35] K. R. Pattipati, Y. Li, and H. A. P. Blom, "A unified framework for the performability evaluation of fault-tolerant computer systems," *IEEE Transactions on Computers*, vol. 42, no. 3, pp. 312–326, 1993.

[36] E. de Souza e Silva and H. R. Gail, "Calculating availability and performability measures of repairable computer systems," *Journal of the ACM*, vol. 36, pp. 171–193, January 1989.

[37] L. Donatiello and V. Grassi, "On evaluating the cumulative performance distribution of fault-tolerant computer systems," *IEEE Transactions on Computers*, vol. 40, no. 11, pp. 1301–1307, 1991.

[38] E. de Souza e Silva and R. Gail, "Calculating transient distributions of cumulative reward," in *Proceedings of Sigmetrics/Performance-95*, (Ottawa, Canada), pp. 231–240, May 1995.

[39] W. H. Sanders, W. D. Obal, M. A. Qureshi, and F. K. Widjanarko, "The *UltraSAN* modeling environment," *Performance Evaluation*, vol. 24, pp. 89–115, 1995.

[40] G. Chiola, M. A. Marsan, G. Balbo, and G. Conte, "Generalized stochastic Petri nets: A definition at the net level and its implications," *IEEE Transactions on Software Engineering*, vol. 19, pp. 89–107, February 1993.

[41] J. Riordan, *An Introduction to Combinatorial Analysis.* John Wiley & Sons, Inc., 1958.

[42] G. M. Constantine, *Combinatorial Theory and Statistical Design.* John Wiley & Sons, Inc., 1987.

[43] D. I. Bailey, "Multiprecision translation and execution of Fortran programs," *ACM Transactions of Mathematical Software*, vol. 19, no. 3, pp. 288–319, 1993.

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

[45] S. M. Ross, *Stochastic Processes.* John Wiley & Sons, 1982.

[46] A. M. Mood and F. A. Graybill, *Introduction to the Theory of Statistics.* McGraw-Hill Book Company, Inc., 1963.

[47] E. de Souza e Silva and R. H. Gail, "An algorithm to calculate transient distributions of cumulative reward," Tech. Rep. CSD-940021, University of California, Los Angeles, May 1994.

[48] H. Weisberg, "The distribution of linear combinations of order statistics from the uniform distribution," *The Annals of Mathematical Statistics*, vol. 42, no. 2, pp. 704–709, 1995.

[49] T. Matsunawa, "The exact and approximate distributions of linear combinations of selected order statistics from a uniform distribution," *The Annals of Institute of Statistical Mathematics*, vol. 37, pp. 1–16, 1985.

[50] S. M. Wilks, *Mathematical Statistics.* John Wiley & Sons, Inc., 1963.

[51] M. A. Qureshi and W. H. Sanders, "Reward model solution methods with impulse and rate rewards: An algorithm and numerical results," *Performance Evaluation*, vol. 20, pp. 413–436, 1994.

[52] LiDIA Manual, "A library for computational number theory," tech. rep., Universitat des Saarlandes, August 1995.

[53] D. Lee, J. Abraham, D. Rennels, and G. Conte, "A numerical technique for the evaluation of large, closed fault-tolerant systems," in *Dependable Computing for Critical Applications*, pp. 95–114, Springer-Verlag, Wien, 1992.

[54] W. H. Sanders and L. M. Malhis, "Dependability evaluation using composed san-based reward models," *Journal of Parallel and Distributed Computing*, vol. 15, no. 3, pp. 238–254, 1992.

[55] R. H. Thomas, "A majority consensus approach to concurrency control for replicated databases," *ACM Transactions on Database Systems*, vol. 4, no. 2, pp. 180–209, 1979.

[56] D. Gifford, "Weighted voting for replicated data," *Proceedings of 7th ACM symposium on Operating Systems Principle*, Pacific Grove, CA, pp. 150–162, 1979.

[57] D. Agarwal and A. E. Abbadi, "The generalized tree protocol: An efficient approach for managing replicated data," *ACM Transactions on Database Systems*, vol. 17, no. 4, pp. 689–717, 1992.

[58] S. Y. Cheung, M. Ahamad, and M. H. Ammar, "Optimizing vote and quorum assignments for reading and writing replicated data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, no. 3, pp. 387–397, 1989.

[59] J. Paris, "Voting with witnesses: A consistency scheme for replicated files," *Proceedings of the 6th Int. Conf. on Data Engineering*, Cambridge, MA, pp. 606–620, 1986.

[60] R. Renesse and A. S. Tanenbaum, "Voting with ghosts," *Proceedings of the 8th IEEE Int. Conf. on Distributed Computing Systems*, pp. 456–462, 1988.

[61] D. Barbara, H. Garcia-Molina, and A. Spauster, "Increasing availability under mutual exclusion constraints with dynamic vote reassignment," *ACM Transactions on Computer Systems*, vol. 7, no. 4, pp. 394–426, 1989.

[62] M. Herlihy, "Dynamic quorum adjustment for partitioned data," *ACM Transactions on Database Systems*, vol. 12, no. 2, pp. 170–194, 1987.

[63] S. Jajodia and D. Mutchler, "Dynamic voting algorithms for maintaining the consistency of a replicated database," *ACM Transactions on Database Systems*, vol. 15, no. 2, pp. 230–280, 1990.

[64] J. Paris and D. D. E. Long, "Efficient dynamic voting algorithms," *Proceedings of 4th IEEE Int. Conf. on Data Engineering*, Los Angeles, CA, pp. 268–275, 1988.

[65] J. Tang and N. Natarajan, "A scheme for maintaining consistency and availability of replicated files in a partitioned distributed system," *Proceedings of the 5th IEEE Int. Conf. on Data Engineering*, pp. 530–537, 1989.

[66] J. C. Laprie, "Dependability: Basic concepts and terminology," *Dependable Computing and Fault-Tolerant Systems*, vol. 5, 1992.

[67] S. E. Butner and R. K. Iyer, "A statistical study of reliability and system load at slac," *Proceedings of the 10th IEEE Int. Symposium on Fault Tolerant Computing*, pp. 207–209, 1980.

[68] X. Castillo and D. P. Siewiorek, "Workload, performance, and reliability of digital computing systems," *Proceedings of the 11th IEEE Int. Symposium on Fault-Tolerant Computing*, pp. 84–89, 1981.

[69] R. K. Iyer, D. J. Rossetti, and M. C. Hseuh, "Measurement and modeling of computer reliability as affected by system activity," *ACM Transactions on Computer Systems*, vol. 4, no. 3, pp. 214–237, 1986.

[70] J. F. Meyer and L. Wei, "Analysis of workload influence on dependability," *Proceedings of the 18th IEEE Int. Symposium on Fault Tolerant Computing*, Tokyo, pp. 84–88, 1988.

[71] M. Ahamad and M. H. Ammar, "Performance characterization of quorum-consensus algorithms for replicated data," *IEEE Transactions on Software Engineering*, vol. 15, no. 4, pp. 492–496, 1989.

[72] S. Y. Cheung, M. H. Ammar, and M. Ahamad, "The grig protocol: A high performance scheme for maintaining replicated data," *Proceedings of the 6th Int. Conf. on Data Engineering*, Cambridge, MA, pp. 438–445, 1990.

[73] A. Kumar, "Performance analysis of a hierarchical quorum consensus algorithm for replicated objects," *Proceedings of the 10th IEEE Int. Conf. on Distributed Computing Systems*, pp. 378–385, 1990.

[74] L. E. Moser, V. Kapur, and P. M. Melliar-Smith, "Probabilistic language analysis of weighted voting algorithms," *Proceedings of the ACM Conf. on Measurement and Modeling of Computer Systems*, pp. 67–73, 1990.

[75] J. B. Dugan and G. Ciardo, "Stochastic Petri net analysis of a replicated file system," *IEEE Transactions on Software Engineering*, vol. 15, no. 4, pp. 394–401, 1989.

[76] R. Geist and K. Trivedi, "Reliability estimation of fault-tolerant systems: Tools and techniques," *Computer*, pp. 52–61, July 1990.

[77] B. R. Haverkort and I. G. Niemegeers, "Performability modelling tools and techniques." To appear in *Performance Evaluation*, 1994.

[78] K. S. Trivedi, B. R. Haverkort, A. Rindos, and V. Mainkar, "Techniques and tools for reliability and performance evaluation: Problems and perspectives," in *Computer Performance Evaluation Modelling Tools and Techniques* (G. Haring and G. Kotsis, eds.), (New York), pp. 1–24, Springer-Verlag, 1994.

[79] C. Lindemann, "DSPNexpress: A software package for the efficient solution of deterministic and stochastic Petri nets," in *Proceedings of the Sixth International Conference on Modelling Techniques and Tools for Computer Systems Performance Evaluation*, (Edinburgh, Great Britain), pp. 15–29, 1992.

[80] M. Bouissou, H. Bouhadana, M. Bannelier, and N. Villatte, "Knowledge modeling and reliability processing: The FIGARO language and associated tools," in *Proceedings of the Twenty-Third Annual International Symposium on Fault-Tolerant Computing*, (Toulouse, France), pp. 680–685, June 1993.

[81] G. Chiola, "A software package for analysis of generalized stochastic Petri net models," in *Proceedings of the International Workshop on Timed Petri Net Models*, (Torino, Italy), pp. 136–143, July 1985.

[82] W. H. Sanders and J. F. Meyer, "METASAN: A performability evaluation tool based on stochastic activity networks," in *Proceedings of the ACM-IEEE Computer Society 1986 Fall Joint Computer Conference*, pp. 807–816, 1986.

[83] G. Florin, P. Lonc, S. Natkin, and J. M. Toudic, "RDPS: A software package for the validation and evaluation of dependable computer systems," in *Proceedings of the Fifth IFAC Workshop Safety of Computer Control Systems (SAFECOMP'86)* (W. J. Quirk, ed.), (Pergamon, Sarlat, France), pp. 165–170, October 1986.

[84] G. Ciardo, J. Muppala, and K. S. Trivedi, "SPNP: Stochastic Petri net package," in *Proceedings of the Fourth International Workshop on Petri Nets and Performance Models*, (Kyoto, Japan), pp. 142–151, December 1989.

[85] C. Béounes, M. Aguéra, J. Arlat, S. Bachmann, C. Bourdeau, J.-E. Doucet, K. Kanoun, J.-C. Laprie, S. Metge, J. M. de Souza, D. Powell, and P. Spiesser, "SURF-2: A program for dependability evaluation of complex hardware and software systems," in *Proceedings of the Twenty-Third International Symposium on Fault-Tolerant Computing*, (Toulouse, France), pp. 668–673, June 1993.

[86] R. German, C. Kelling, A. Zimmermann, and G. Hommel, "Timenet: A toolkit for evaluating non-Markovian stochastic Petri nets," tech. rep., Technische Universitat Berlin, Germany, 1994. Submitted for publication.

[87] J. Couvillion, R. Freire, R. Johnson, W. D. Obal II, M. A. Qureshi, M. Rai, W. H. Sanders, and J. Tvedt, "Performability modeling with *UltraSAN*," *IEEE Software*, vol. 8, pp. 69–80, September 1991.

[88] M. Siegle, "Reduced Markov models of parallel programs with replicated processes," in *Proceedings of the Second Euromicro Workshop on Parallel and Distributed Processing*, (Malaga), pp. 1–8, January 1994.

[89] W. H. Sanders and R. S. Freire, "Efficient simulation of hierarchical stochastic activity network models," *Discrete Event Dynamic Systems: Theory and Applications*, vol. 3, pp. 271–300, July 1993.

[90] W. D. Obal II and W. H. Sanders, "Importance sampling simulation in *UltraSAN*," *Simulation*, vol. 62, pp. 98–111, February 1994.

[91] W. D. Obal II and W. H. Sanders, "An environment for importance sampling based on stochastic activity networks," in *Proceedings of the 13th Symposium on Reliable Distributed Systems*, (Dana Point, California), pp. 64–73, October 1994.

[92] Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, University of Illinois, *UltraSAN Reference Manual*, 1995.

[93] A. Movaghar and J. F. Meyer, "Performability modeling with stochastic activity networks," in *Proceedings of 1984 Real-Time Systems Symposium*, (Austin, Texas), pp. 215–224, December 1984.

[94] E. E. Lewis and F. Böhm, "Monte Carlo simulation of Markov unreliability models," *Nuclear Engineering and Design*, vol. 77, pp. 49–62, January 1984.

[95] V. F. Nicola, P. Heidelberger, and P. Shahabuddin, "Uniformization and exponential transformation: Techniques for fast simulation of highly dependable non-Markovian systems," in *Proceedings of the Twenty-Second Annual International Symposium on Fault-Tolerant Computing*, (Boston, Massachusetts), pp. 130–139, 1992.

[96] A. M. Blum, P. Heidelberger, S. S. Lavenberg, M. Nakayama, and P. Shahabuddin, "System availability estimator (SAVE) language reference and user's manual version 4.0," Tech. Rep. RA 219 S, IBM Thomas J. Watson Research Center, June 1993.

[97] P. Shahabuddin, *Simulation and Analysis of Highly Reliable Systems.* PhD thesis, Stanford University, 1990.

[98] B. P. Shah, "Analytic solution of stochastic activity networks with exponential and deterministic activities," Master's thesis, The University of Arizona, 1993.

[99] J. E. Tvedt, "Matrix representations and analytical solution methods for stochastic activity networks," Master's thesis, The University of Arizona, 1990.

[100] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 1988.

[101] O. Osterby and Z. Zlatev, *Lecture Notes in Computer Science: Direct Methods for Sparse Matrices.* Heidelberg: Springer-Verlag, 1983.

[102] U. R. Krieger, B. Müller-Clostermann, and M. Sczittnick, "Modeling and analysis of communication systems based on computational methods for Markov chains," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 9, pp. 1630–1648, 1990.

[103] C. Lindemann, "An improved numerical algorithm for calculating steady-state solutions of deterministic and stochastic Petri net models," in *Proceedings of the Fourth International Workshop on Petri Nets and Performance Models*, (Melbourne, Australia), pp. 176–185, 1991.

[104] M. A. Marsan and G. Chiola, "On Petri nets with deterministic and exponentially distributed firing times," in *Advances in Petri Nets 1986* (G. Rozenberg, ed.), Lecture Notes in Computer Science 266, pp. 132–145, Springer, 1987.

[105] B. L. Fox and P. W. Glynn, "Computing Poisson probabilities," *Communications of the ACM*, vol. 31, pp. 440–445, 1988.

[106] V. F. Nicola, M. K. Nakayama, P. Heidelberger, and A. Goyal, "Fast simulation of dependability models with general failure, repair and maintenance processes," in *Proceedings of the Twentieth Annual International Symposium on Fault-Tolerant Computing*, (Newcastle upon Tyne, United Kingdom), pp. 491–498, June 1990.

[107] L. M. Malhis, W. H. Sanders, and R. D. Schlichting, "Analytic performability evaluation of a group-oriented multicast protocol," Tech. Rep. PMRL 93-13, The University of Arizona, June 1993. Submitted for publication.

[108] B. D. McLeod and W. H. Sanders, "Performance evaluation of N-processor time warp using stochastic activity networks," Tech. Rep. PMRL 93-7, The University of Arizona, March 1993. Submitted for publication.

[109] W. H. Sanders, L. A. Kant, and A. Kudrimoti, "A modular method for evaluating the performance of picture archiving and communication systems," *Journal of Digital Imaging*, vol. 6, pp. 172–193, August 1993.