EMPIRICAL COMPARISON OF UNIFORMIZATION METHODS FOR CONTINUOUS-TIME MARKOV CHAINS

by

John D. Diener

A Thesis Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

MASTER OF SCIENCE
WITH A MAJOR IN ELECTRICAL ENGINEERING

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 9 4

STATEMENT BY AUTHOR

SIGNED:____________________

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

____________________ ____________

William H. Sanders
Associate Professor of
Electrical and Computer Engineering

Date

To my family, and especially Anna,

for their continuous support and belief in me.

ACKNOWLEDGMENTS

There are a number of people to whom I owe thanks for their help and support during work on this thesis. First, I thank my graduate committee members, Dr. Michael Marcellin, Dr. Aad P.A. van Moorsel, and my advisor Dr. William Sanders, for reviewing my thesis. Bill deserves special thanks for his guidance throughout the project.

I would also like to thank the PMRL group for their help and friendship. In particular, thanks to Anand Kuratti for helping me through the complexities of the thesis process, and Aad for his thorough readings and helpful discussions.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Computation of transient state occupancy probabilities of continuous-time Markov chains is important for evaluating many dependability and performability models. Numerical methods have been developed to perform this computation, including ordinary differential equation solution methods and uniformization, however, their performance degrades with increasing model stiffness. A newly proposed variant of uniformization, adaptive uniformization (AU), can potentially avoid such degradation. This thesis presents results of three AU implementations, differing in the method by which the "jump probabilities" are calculated. Evaluation of these methods is carried out through use of a *C++* class that computes a bound on the round-off error incurred, and counts the number of arithmetic instructions performed. Several example models illustrate use of the class and compare the adaptive uniformization implementations with standard uniformization. Results show that for certain models and mission times, adaptive uniformization can realize significant efficiency gains over standard uniformization, while maintaining the stability of standard uniformization.

CHAPTER 1

INTRODUCTION

Continuous-time Markov chains are often used to evaluate the performance, dependability, and performability of computer systems and networks. For many models [6, 7], transient measures are of particular interest. Since for large problems, closed-form solution is infeasible, numerical approaches are used. These include ordinary differential equation methods such as Runge-Kutta-Fehlberg or an implicit method TR-BDF2 [24], Jensen's method [11] (commonly known as *uniformization*), and combinations thereof [15].

Uniformization has often been considered the best method [9, 10, 25] for computing transient state probabilities of continuous-time Markov chains, but suffers a dramatic degradation in performance as the highest state departure rate increases for any fixed time point [15]. For these so-called "stiff models," the truncation point needed to achieve acceptable accuracy becomes prohibitively large. *Adaptive uniformization* (AU) [18] was introduced to avoid this problem in models where states with high transition rates are not reached for several state transitions, and is particularly applicable for models with short mission times, where ordinary differential equation methods are inefficient. Additionally, AU shares a nice feature with uniformization in that an error bound can be prespecified, and computes an absolute lower bound result within the specified error. Adaptive uniformization takes advantage of the fact that there is typically a single initial state, and several transitions are necessary until all states are reachable, or *active*. Thus the uniformization rate at each step need only be greater than or equal to the largest outgoing rate from the set of currently active states, not greater than the largest rate in the entire state space. This allows bigger *jumps* to be made, reducing the number of iterations necessary to reach a solution.

While adaptive uniformization will always take a number of iterations less than or equal to standard uniformization to achieve a specified level of accuracy, most iterations require more computation, so it is not clear when AU will be computationally advantageous. Theoretical results [18] suggest that there are situations when this will be the case, but no implementations have been made that show that this is indeed true, or the magnitude of savings (or expenses) that may be incurred. In this thesis, results of three different implementations of adaptive uniformization are presented, and are compared with regular uniformization. The three implementations differ in how the "jump probabilities" are calculated, the most critical part of the adaptive uniformization algorithm. In order to make a careful assessment of the various implementations, a *C++* class was developed to instrument the code. This new class calculates a bound on the round-off error incurred by each method, and the number of floating point operations required for solution of a model. The number of operations is broken down by operation type (i.e., multiplication, division, addition, and subtraction) and by the portion of the algorithm the operations are associated with. The fine-grained nature of this data collection permits a clear understanding of the exact costs of the different portions of each algorithm, as well as when it will encounter numerical difficulties.

The objectives of this thesis are three-fold. First, three AU algorithms are implemented to be compatible with *UltraSAN* [2], a software package with which it is possible to model systems graphically as stochastic activity networks (SANs) [20]. From this graphical description, *UltraSAN* is capable of generating a CTMC and the associated reward structure. However, this thesis focuses only on the resulting CTMC and its transient solution. Second, it intends to show that it is indeed possible to instrument such algorithms to gain a clear understanding of exact costs and numerical issues of different algorithms. The *C++* class developed to do this is algorithm independent, and can be easily added to any algorithm implemented in *C* or *C++*. Finally, given the above, this thesis provides an accurate comparison, using the instrumented code, of the three adaptive uniformization implementations and standard uniformization in the context of several example models. The results show the advantages/disadvantages of the new approach are clearly problem dependent, but that there are realistic situations where adaptive uniformization is dramatically more efficient than SU, while retaining the stability of standard uniformization.

The thesis is organized as follows. After the theoretical background of uniformization and adaptive uniformization is reviewed in the next chapter, the details of computing the jump probabilities for AU are given in Chapter 3. Three methods, ACE [23], a modified ACE algorithm [18] and uniformization, are discussed in the context of the implementations. This chapter also explains modifications to the Fox and Glynn Poisson probability calculation, and compares and describes the overall algorithm implementation. Chapter 4 explains the *C++* class for evaluating the viability of the methods and results. Chapter 5 details results based upon instrumentation, empirically comparing uniformization and adaptive uniformization. Finally, Chapter 6 summarizes the thesis results and presents ideas for further research.

CHAPTER 2

BACKGROUND

This chapter explains the theory behind the methods used in this thesis to solve for the transient state probabilities of continuous-time Markov chains (CTMCs). Let CTMC $Y = \{Y(t); t \geq 0\}$ be defined on the state space S where it is assumed $S = \{0, 1, 2, \ldots\}$. Let $Q = (q(i,j)), i, j \in S$, be the infinitesimal generator matrix of Y, with $q_i = -q(i,i)$ for $i \in S$. The row vector $\underline{\pi}(0) = (\pi_0(0), \pi_1(0), \ldots)$ denotes the initial probability distribution of S . The objective is then to obtain the transient state probability vector $\underline{\pi}(t)$, with $\pi_i(t) = \text{Prob}\{Y(t) = i\}$ for any $i \in S$. Only numerical techniques will be discussed, since closed-form solution for models having over 400 states is extremely difficult [32].

Kolmogorov's differential equations [4] show that $\underline{\pi}(t)$ follows

$$\frac{d\underline{\pi}(t)}{dt} = \underline{\pi}(t)Q, \text{ given } \underline{\pi}(0). \tag{2.1}$$

Therefore, this problem has the solution $\underline{\pi}(t) = \underline{\pi}(0)e^{Qt}$. However, directly solving e^{Qt} does not lead to satisfactory numerical algorithms [8, 16].

2.1 Standard Uniformization

In standard uniformization (SU), which is often used in the context of Markov models, the CTMC is decomposed into a discrete time Markov chain (DTMC) and a Poisson process [1]. For later use, standard uniformization is briefly reviewed. Let $\lambda \geq \max\{q_i\}, i \in S$, be the so-called *uniformization rate.* Then, the DTMC $X = \{X_n, n = 0, 1, \ldots\}$, can be defined, with transition matrix $P = I + (1/\lambda)Q$. $\underline{\pi}(t)$ is then obtained from

$$\underline{\pi}(t) = \underline{\pi}(0) \sum_{n=0}^{\infty} e^{-\lambda t} \frac{(\lambda t)^n}{n!} P^n. \tag{2.2}$$

By introducing $\underline{\pi}_n = \underline{\pi}_{n-1} P, n = 1, 2, \ldots,$ where $\underline{\pi}_0 = \underline{\pi}(0)$, Equation 2.2 can be written recursively as

$$\underline{\pi}(t) = \sum_{n=0}^{\infty} U_n(t) \underline{\pi}_n \tag{2.3}$$

where for later use the *jump probabilities* $U_n(t)$ have been defined as

$$U_n(t) = e^{-\lambda t} \frac{(\lambda t)^n}{n!}. \tag{2.4}$$

$\underline{\pi}_n$ is interpreted as the state probability distribution of X after n state transitions. Thus $\underline{\pi}(t)$ can be thought of as the discrete probability vector after n jumps multiplied by the probability of having n transitions occur, summed over all possible number of jumps. In SU, these jump probabilities form a Poisson distribution. Notice the recursive definition of Equation 2.3 allows the use of sparse matrix techniques for Q and P. In this implementation of SU, the infinite sum of Equation 2.3 is both left and right truncated so that

$$\underline{\pi}(t) \approx \sum_{n=N_{sl}}^{N_{sr}} e^{-\lambda t} \frac{(\lambda t)^n}{n!} \underline{\pi}_n, \tag{2.5}$$

and

$$\sum_{n=0}^{N_{sl}-1} e^{-\lambda t}\frac{(\lambda t)^n}{n!} + \sum_{n=N_{sr}+1}^{\infty} e^{-\lambda t}\frac{(\lambda t)^n}{n!} < \epsilon. \tag{2.6}$$

N_{sl} and N_{sr} are then selected so that the resulting *truncation error* is less than an $\epsilon > 0$. Notice that this left truncation does not effect the number of matrix-vector multiplications that must occur and only reduces the amount of summation necessary. It can then be seen, with $\|\underline{x}\|_\infty = \max\{|x_i|\}$, that

$$\left\| \left(\sum_{n=0}^{N_{sl}-1} e^{-\lambda t}\frac{(\lambda t)^n}{n!} + \sum_{n=N_{sr}+1}^{\infty} e^{-\lambda t}\frac{(\lambda t)^n}{n!} \right) \underline{\pi}_n \right\|_\infty \le \sum_{n=0}^{N_{sl}-1} e^{-\lambda t}\frac{(\lambda t)^n}{n!} + \sum_{n=N_{sr}+1}^{\infty} e^{-\lambda t}\frac{(\lambda t)^n}{n!} < \epsilon \tag{2.7}$$

since all elements of $\underline{\pi}_n$ are in $[0, 1]$. Fox and Glynn [5] have given a method to find N_{sl} and N_{sr} given a predefined ϵ, λ, and t, and using this, SU allows solution of $\underline{\pi}(t)$ to any desired accuracy $\epsilon > 0$.

2.2 Adaptive Uniformization

In adaptive uniformization, the uniformization rate can change with the number of jumps, or current *epoch*, and depends on the set of states the DTMC X can be in at some epoch. The set of *active states at epoch* $n, n = 0, 1, \ldots,$ of X is defined as $A_n \subseteq S$ where

$$A_n = \{i \in S \mid \underline{\pi}_n(i) > 0\}. \tag{2.8}$$

Then the *adapted uniformization rates* can be defined as $\lambda_n \ge \max\{q_i \mid i \in A_n\}$, for $n = 0, 1, \ldots$. Restricted state-transition rate matrices for each step n, called *adapted* *infinitesimal generators* in [18], can be defined as the $Q_n = (q_n(i,j)), i,j \in S$, such that

$$q_n(i,j) = \begin{cases} q(i,j) & \text{if } i \in A_n \\ 0 & \text{otherwise} \end{cases}$$

Only rows of Q corresponding to active states in each epoch are considered. Next, the *adapted transition matrices* are given by

$$P_n = I + \frac{1}{\lambda_n} Q_n, n = 0, 1, \ldots. \tag{2.9}$$

If $U_n(t)$ is defined to be the probability of n jumps in time t in the birth process formed by the adaptive uniformization rates $\lambda_0, \lambda_1, \ldots$, then the transient state probability vector can be defined as

$$\underline{\pi}(t) = \underline{\pi}(0) \sum_{n=0}^{\infty} U_n(t) \prod_{i=0}^{n-1} P_i = \sum_{n=0}^{\infty} U_n(t) \underline{\pi}_n, \tag{2.10}$$

where

$$\underline{\pi}_n = \underline{\pi}_{n-1} P_{n-1}, n = 1, 2, \ldots \text{ with } \underline{\pi}_0 = \underline{\pi}(0). \tag{2.11}$$

As with SU, the sum is truncated so that

$$\underline{\pi}(t) \approx \sum_{n=0}^{N_a} U_n(t) \underline{\pi}_n \text{ with } \sum_{n=N_a+1}^{\infty} U_n(t) < \epsilon. \tag{2.12}$$

Left truncation is not employed since the jump probabilities no longer form a Poisson distribution. Like SU, AU solutions can be found to any accuracy $\epsilon > 0$. Since all elements of $\underline{\pi}_n$ are in $[0,1]$,

$$\left\| \sum_{n=N_a+1}^{\infty} U_n(t) \underline{\pi}_n \right\|_{\infty} \leq \sum_{n=N_a+1}^{\infty} U_n(t) < \epsilon. \tag{2.13}$$

By using a pure birth process to find the jump probabilities, it is possible to make probabilistically larger jumps with AU than SU. Thus, for identical ϵ and t, $N_a \leq N_{sr}$, potentially resulting in a reduction of computation time.

CHAPTER 3

IMPLEMENTATION DETAILS

This chapter begins with a discussion about the main difficulty of adaptive uniformization, the calculation of the jump probabilities. The three methods implemented in this study are examined first. A short explanation of the Poisson probability calculation follows while a look at the overall algorithm implementation concludes the chapter.

3.1 Jump Probability Calculation

The basic difference between AU and SU lies in calculation of the jump probabilities. For SU, the jump process has a Poisson distribution, while AU's takes the form of a general birth process. It can be defined by $B = \{B(t); t \geq 0\}$ over the state space $S = \{0, 1, \ldots\}$. Then $U_n(t) = \text{Prob}\{B(t) = n\}$, with B's infinitesimal generator matrix Q_B of the form

$$Q_B = \begin{pmatrix} -\lambda_0 & \lambda_0 & 0 & \ldots & \ldots & 0 & \ldots & \ldots \\ 0 & -\lambda_1 & \lambda_1 & 0 & \ldots & 0 & \ldots & \ldots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \ldots & \ldots & \ldots & 0 & -\lambda_{N_a} & \lambda_{N_a} & \ldots \\ 0 & \ldots & \ldots & \ldots & \ldots & 0 & -\lambda_{N_a+1} & \ldots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \tag{3.1}$$

where λ_i is as defined in Section 2.2. For practical purposes, the infinite matrix Q_B is of course truncated to the state space $S = \{0, 1, \ldots, N_a\}$. In the following sections, the three alternatives used in this study to compute the jump probabilities for AU, an acyclic Markov chain evaluator (ACE) developed by [28] then refined in [23], a modified ACE scheme from [18], and standard uniformization, will be discussed. Special attention will be given to issues related to their implementation in the context of having no prior knowledge of the adaptive uniformization rates.

3.1.1 ACE

ACE is applicable to solving this problem both because of B's acyclic nature and that the ACE algorithm is recursive. The method is based upon the fact that

$$U_n(t) = \pi_{B,i}(t) = \sum_{\gamma_j \in \Gamma} \sum_{k=0}^{K_i(\gamma_j)} a_{i,j,k} t^k e^{\gamma_j t}, \text{ when } n = i \tag{3.2}$$

is the general form of the transient solution of a CTMC, where Γ is the set of unique poles and $K_i(\gamma_j), \gamma_j \in \Gamma$ is the number of poles with value γ_j in the Laplace transform expression [23]. The a coefficients are indexed by i, j, and k, representing the ith state, jth unique pole, and $(k+1)$th appearance of the jth pole. Thus γ_j is the jth unique pole. Calculation of $a_{i,j,k}$ can be accomplished recursively through the method of [23] in the special case of an acyclic CTMC. In this study numerical improvements shown in [13] and [14] to reduce cancellation error and chance of overflow have been incorporated. First, the generator matrix $Q_B = (q_B(i,j)), i, j \in S$, is scaled by the mission time t, resulting in

$$\overline{Q}_B = tQ_B, \; \overline{t} = 1.0, \; \overline{\gamma}_j = t\gamma_j \tag{3.3}$$

and thus

$$\overline{\lambda}_i = t\lambda_i = -\overline{q}_B(i,i), \text{ for } i = 0, 1, \ldots, N_a. \tag{3.4}$$

Second, we have for the initial state 0

$$\overline{\gamma}_1 = -\overline{\lambda}_0,\ K_0(\overline{\gamma}_1) = 0,\ \overline{a}_{0,1,0} = e^{-\overline{\lambda}_0}. \tag{3.5}$$

Finally, the other coefficients are derived recursively by the following scheme starting with state 1

$$\overline{a}_{i,j,k} = \begin{cases} \overline{a}_{i-1,j,k}\frac{\overline{\lambda}_{i-1}}{\overline{\gamma}_j+\overline{\lambda}_i} & \text{iff } \overline{\lambda}_i \neq \overline{\gamma}_j \text{ and } k = K_i(\overline{\gamma}_j) \quad (a) \\ \overline{a}_{i-1,j,k}\frac{\overline{\lambda}_{i-1}}{\overline{\gamma}_j+\overline{\lambda}_i} - (k+1)\frac{\overline{a}_{i,j,k+1}}{\overline{\gamma}_j+\overline{\lambda}_i} & \text{iff } \overline{\lambda}_i \neq \overline{\gamma}_j \text{ and } 0 \leq k < K_i(\overline{\gamma}_j) \quad (b) \\ \overline{a}_{i-1,j,k-1}\frac{\overline{\lambda}_{i-1}}{k} & \text{iff } \overline{\lambda}_i = \overline{\gamma}_j \text{ and } 1 \leq k \leq K_i(\overline{\gamma}_j) \quad (c) \\ -\sum_{\overline{\gamma}_m \in \overline{\Gamma}, \overline{\gamma}_m \neq -\overline{\lambda}_i} \overline{a}_{i,m,0} e^{-\overline{\lambda}_i - \overline{\gamma}_m} & \text{otherwise} \quad (d) \end{cases} \tag{3.6}$$

Given these improvements from [13], computing $\pi_{B,i}(t)$ from the ACE coefficients is a simple matter of summation, so

$$\pi_{B,i}(t) = \sum_{\overline{\gamma}_j \in \overline{\Gamma}} \sum_{k=0}^{K_i(\overline{\gamma}_j)} \overline{a}_{i,j,k}. \tag{3.7}$$

In the implementation the sum of Equation 3.6 (d) is calculated such that positive and negative terms are added in ascending order according to their absolute values, with the addition of these partial results being the total sum, thereby reducing cancellation error [13]. This scheme is not employed in the $\pi_{B,i}(t)$ summation. The additional overhead is high because of ACE's complexity, and more importantly, it would not contribute to the stability of the method. Since the coefficients for the current state's transient probability are only dependent upon the previous state's, ACE for this specialized type of acyclic CTMC can be implemented dynamically, with only two sets of coefficients held in memory simultaneously. However, each new iteration adds another coefficient, making ACE an $O(N_a^2)$ algorithm.

3.1.2 Modified ACE

To reduce the computational complexity from $O(N_a^2)$ to $O(N_a)$, [18] introduced a modified version of ACE for when λ_i converges to a fixed rate λ. Convergence is typical behavior in AU and is also a by-product of all states becoming active. Although it is not necessarily the case if strict rules of adaptive uniformization are followed, this implementation designates the maximum rate λ to be the converged rate. As soon as the state with rate λ becomes active, the birth process is assumed to have converged. As will be seen, this notion of convergence is important to making adaptive uniformization computationally beneficial.

So, if convergence occurs at epoch m, it can be said $\lambda_{m+l} = \lambda$ for $l = 0, 1, \ldots$, implying that the birth process B behaves as a Poisson process from epoch m onwards. By convolving a hypo-exponential and Poisson distribution [18], and taking into account scaling by t, an equation for $U_{m+l}(t)$ can be derived:

$$\begin{aligned} U_{m+l}(t) \quad &= \sum_{\overline{\gamma}_j \in \overline{\Gamma}, \overline{\gamma}_j \neq -\overline{\lambda}} \sum_{k=0}^{K_m(\overline{\gamma}_j)} \sum_{r=0}^{k} \overline{b}_{m,j,k} (-1)^r \frac{k!}{r!(k-r)!} \frac{\overline{\lambda}^l}{(\overline{\lambda} + \overline{\gamma}_j)^{r+l+1}} \frac{(r+l)!}{l!} e^{\overline{\gamma}_j} \\ - &\sum_{\overline{\gamma}_j \in \overline{\Gamma}, \overline{\gamma}_j \neq -\overline{\lambda}} \sum_{k=0}^{K_m(\overline{\gamma}_j)} \sum_{r=0}^{k} \overline{b}_{m,j,k} (-1)^r \frac{k!}{r!(k-r)!} \frac{\overline{\lambda}^l}{(\overline{\lambda} + \overline{\gamma}_j)^{r+l+1}} \frac{(r+l)!}{l!} e^{-\overline{\lambda}} \sum_{v=0}^{r+l} \frac{(\overline{\lambda} + \overline{\gamma}_j)^v}{v!}, \end{aligned} \tag{3.8}$$

given that $\overline{\lambda} \neq \overline{\lambda}_i$ for $0 \leq i < m$. By creating terms A, C and D, $U_{m+l}(t)$ can be computed recursively, with a constant number of operations per step. So, let

$$A_{j,k,r}^{(\zeta)}(l) = \overline{b}_{m,j,k} (-1)^r \frac{k!}{r!(k-r)!} \frac{\overline{\lambda}^l}{(\overline{\lambda} + \overline{\gamma}_j)^{r+l+1}} \frac{(r+l)!}{l!} e^{\zeta} \tag{3.9}$$

$$C_{j,r}(l) = \sum_{v=0}^{r+l} \frac{(\overline{\lambda} + \overline{\gamma}_j)^v}{v!} \tag{3.10}$$

for $l \geq 0$. Recursive relations, necessary for implementation, for $l > 0$ are

$$A^{(\zeta)}_{j,k,r}(l) = A^{(\zeta)}_{j,k,r}(l-1) \frac{\overline{\lambda}}{(\overline{\lambda} + \overline{\gamma}_j)} \frac{(r+l)}{l} \tag{3.11}$$

$$C_{j,r}(l) = C_{j,r}(l-1) + D_{j,r}(l) \tag{3.12}$$

where

$$D_{j,r}(l) = \frac{(\overline{\lambda} + \overline{\gamma}_j)^{r+l}}{(r+l)!} = D_{j,r}(l-1) \frac{(\overline{\lambda} + \overline{\gamma}_j)}{r+l}. \tag{3.13}$$

Therefore, $U_{m+l}(t)$ can be expressed more succinctly as

$$U_{m+l}(t) = \sum_{\overline{\gamma}_j \in \overline{\Gamma}, \overline{\gamma}_j \neq -\overline{\lambda}} \sum_{k=0}^{K_m(\overline{\gamma}_j)} \sum_{r=0}^{k} \left[A^{(\overline{\gamma}_j)}_{j,k,r}(l) - A^{(-\overline{\lambda})}_{j,k,r}(l) C_{j,r}(l) \right]. \tag{3.14}$$

In the implementation, for $i \leq m$, $U_i(t)$ is found via ACE. When the converged rate λ is encountered, values for $A^{(\zeta)}_{j,k,r}(0)$, $C_{j,r}(0)$ and $D_{j,r}(0)$ are calculated. The conversion [19] from $\overline{a}_{m,j,k}$ in Equation 3.6 to $\overline{b}_{m,j,k}$ in Equation 3.8 is given by

$$\begin{aligned}
\overline{b}_{m,j,0} &= (\overline{\lambda} + \overline{\gamma}_j)\overline{a}_{m,j,0} \\
\overline{b}_{m,j,k} &= (\overline{\lambda} + \overline{\gamma}_j)\overline{a}_{m,j,k} + (k+1)\overline{a}_{m,j,k+1} \\
\overline{b}_{m,j,K_m(\overline{\gamma}_j)} &= (\overline{\lambda} + \overline{\gamma}_j)\overline{a}_{m,j,K_m(\overline{\gamma}_j)}
\end{aligned} \tag{3.15}$$

Then $U_{m+l}(t)$ for $l > 0$ is found from Equation 3.14 and by updating the A, C and D coefficients via Equations 3.11, 3.12 and 3.13. As can be seen, from this point onwards the computational complexity is linear, and thus Modified ACE is an $O(N_a)$ algorithm. Further simplification is possible when $\lambda_i \neq \lambda_j, i \neq j$ for all $i, j \in \{0, \ldots, m\}$. Then

$K_m(\overline{\gamma}_j) = 0$ for all $\overline{\gamma}_j$, and the sums over k and r disappear, thus requiring less calculation per epoch. It then holds that

$$A_j^{(\zeta)}(l) = \overline{b}_{m,j,0} \frac{\overline{\lambda}^l}{(\overline{\lambda}+\overline{\gamma}_j)^{l+1}} e^{\zeta} = \frac{\overline{\lambda}}{(\overline{\lambda}+\overline{\gamma}_j)} A_j^{(\zeta)}(l-1) \tag{3.16}$$

$$C_j(l) = \sum_{v=0}^{l} \frac{(\overline{\lambda}+\overline{\gamma}_j)^v}{v!} = C_j(l-1) + D_j(l) \tag{3.17}$$

with

$$D_j(l) = \frac{(\overline{\lambda}+\overline{\gamma}_j)^l}{l!} = \frac{(\overline{\lambda}+\overline{\gamma}_j)}{l} D_j(l-1). \tag{3.18}$$

Finally, this gives

$$U_{m+l}(t) = \sum_{\overline{\gamma}_j \in \overline{\Gamma}, \overline{\gamma}_j \neq -\overline{\lambda}} \left[A_j^{(\overline{\gamma}_j)}(l) - A_j^{(-\overline{\lambda})}(l) C_j(l) \right]. \tag{3.19}$$

The summation for $U_{m+l}(t)$ in Equation 3.14 or 3.19 is done by adding the terms in ascending order of their magnitude, for both positive and negative numbers, then summing these two intermediary results. This reduces cancellation error, and is reasonable to implement within modified ACE because it is of order $O(N_a)$. Since the coefficients A, C and D are independent of one another, all terms can be held in a single array that is updated each jump.

3.1.3 Uniformization

The final method used to compute jump probabilities is standard uniformization itself. The variant of AU that uses this is called *layered uniformization* (LU) [18]. Let $\lambda = \max\{\lambda_i\}, i = 0, \ldots, N_a$. Then

$$U_n(t) = \sum_{k=N_{Bl}}^{N_{Br}} e^{-\lambda t} \frac{(\lambda t)^k}{k!} \underline{\pi}_k^B(n), \text{ for } n = 0, \ldots, N_a \tag{3.20}$$

where

$$\underline{\pi}_k^B = \underline{\pi}_{k-1}^B P_B, \text{ with } \underline{\pi}_0^B(0) = 1 \tag{3.21}$$

and

$$P_B = I + \frac{1}{\lambda} Q_B. \tag{3.22}$$

As shown in Section 2, the truncation bounds N_{Bl} and N_{Br} can be selected such that the resulting truncation error is smaller than an $\epsilon_B > 0$. The bound on the total error from truncation in LU is then given by $\epsilon + \epsilon_B$ [17].

In the actual implementation, Q_B is found a step at a time based on the set of active states, which, due to its special nature, can be stored as an array rather than a sparse matrix. Thus there exists a P_{B_n} rather than a P_B, for each n such that

$$P_{B_n} = I + \frac{1}{\lambda_k} Q_{B_n} \text{ with } \lambda_k = \max\{\lambda_i\}, i = 0, \ldots, n. \tag{3.23}$$

So, each state (jump) probability is calculated separately, as each new adapted uniformization rate is found. As with modified ACE, reaching the converged rate λ at some epoch m is an important step. Then it is assumed that $\lambda_i = \lambda$ for $i = m, \ldots, N_a, \ldots$ and all the rest of the jump probabilities can be found immediately via the standard uniformization in Equation 3.20. Since the converged rate is taken to be λ, the maximum rate in the CTMC Y, the vector matrix multiplication in Equation 3.21 reduces to $\underline{\pi}_k^B(n) = \underline{\pi}_{k-1}^B(n-1)$ for $n > m$. This combined with using the Fox and Glynn algorithm results in a total order of complexity of $O(N_a\sqrt{N_{Br}})$ [17]. Notice that *a priori* knowledge of all adaptive uniformization rates would reduce the total computation necessary, but not the level of complexity.

3.1.4 Summary of Methods

The three alternatives used in this study for calculation of the jump probabilities for adaptive uniformization have been discussed. Implementations for all three options were described in the context of dynamic calculation of adaptive uniformization rates, that is, new rates are found per iteration as opposed to knowing them all in advance. The issue of reducing error in ACE and Modified ACE was also addressed. Since the coefficients in these algorithms are unbounded and possibly negative, these two methods are potentially unstable. In fact, both implementations have instability checks that cause termination of the transient calculation whether accuracy demands have been met or not. However, testing is required to effectively gauge their instability. On the other hand, uniformization is inherently stable because it uses only positive, bounded elements. In terms of complexity, Modified ACE is best, followed by uniformization and then ACE. Experimental results will be given in Section 5 to compare these methods' usefulness. Table 3.1 summarizes these three methods and the jump probability calculation for SU in terms of complexity and potential stability.

Technique	*Complexity*	*Stability*
ACE	$O(N_a^2)$	Unstable
Modified ACE	$O(N_a)$	Unstable
Uniformization (LU)	$O(N_a\sqrt{N_{Br}})$	Stable
Poisson Probabilities (SU)	$O(\sqrt{N_{sr}})$	Stable

Table 3.1: Complexity and Stability of Methods

3.2 Poisson Probability Calculation

For both SU and LU, Poisson probabilities are calculated using the method of Fox and Glynn [5]. Starting with the mode, which is assigned a weight with the intent to avoid underflow, the remaining weights between the left and right truncation points are found recursively. These weights are then divided by a constant α to approximate the probabilities. In [5], $\alpha \doteq W$ where W is the sum off all weights within the truncation points. The result of this is that the truncated probabilities always sum to 1, and thus they err on the high side. Thus, transient state probabilities computed via SU or LU cannot be said to be an absolute lower or upper bound on the actual result.

To create a more rigorous bound, the Fox and Glynn method has been modified slightly. α is redefined such that

$$\alpha = W(1 + \epsilon_{LR}), \tag{3.24}$$

and define the tail masses

$$T(i) = \sum_{j=0}^{i} p(j), \quad Q(i) = \sum_{j=i}^{\infty} p(j). \tag{3.25}$$

If L and R are the truncation points, λ the Poisson rate, and $m = \lfloor \lambda \rfloor$ is the mode, then ϵ_{LR} is ideally given by the bounds on the tail masses as

$$\epsilon_{LR} = T(L-1) + Q(R+1). \tag{3.26}$$

However, T and Q cannot be determined exactly, but are bounded in [5] such that

$$T(L-1) \leq (1 + 1/\lambda)e^{1/8\lambda} \int_{-\infty}^{(m-L-1/2)/\sqrt{\lambda}} \frac{1}{\sqrt{2\pi}} e^{-s^2/2} ds \tag{3.27}$$

and

$$Q(R+1) \le \frac{(1+1/\lambda)e^{1/16}\sqrt{2}}{1-e^{-2(R-m+1)/9}} \int_{-\infty}^{(R-m-1/2)/\sqrt{2\lambda}} \frac{1}{\sqrt{2}\pi} e^{-s^2/2} ds. \tag{3.28}$$

In the implementation, the above integrals are themselves bounded by

$$\int_{-\infty}^{x} \frac{1}{\sqrt{2}\pi} e^{-s^2/2} ds \le \frac{(1/\sqrt{2}\pi)e^{-x^2/2}}{x}, \tag{3.29}$$

with error less than $\{(1/\sqrt{2}\pi)e^{-x^2/2}\}/x^3$ for $x > 0$, in order to simplify calculation. So, if ϵ_{LR} is equated to the sum of these bounds, α becomes an upper bound on the summation of the weights. Hence, dividing by α to find the probabilities gives lower bound results both for the Poisson probabilities and the transient state probabilities.

3.3 Algorithm Implementation

Given these methods to compute the jump probabilities, this section outlines the algorithms used in the four uniformization methods considered. Inputs to the algorithms are t, ϵ, the infinitesimal generator matrix Q of the CTMC, and the initial state v. LU also requires a value for ϵ_B. Both the SU and AU algorithms are given in Figure 3.1, with parts common to both methods centered. Notice that adaptive uniformization varies slightly depending upon which method is chosen to calculate the jump probabilities.

In both SU and AU, Q is read into a sparse matrix structure, holding only the non-zero elements, with λ found during read-in. In SU, P is calculated immediately, along with the jump probabilities $U_{N_{sl}}(t), \ldots, U_{N_{sr}}(t)$. On the other hand, AU calculates $U_n(t)$ on a per epoch basis from Equations 3.2, 3.14 or 3.20, except for when the converged rate is reached in LU. In that case all remaining probabilities are calculated directly from

<table>
<tr><td>Standard Uniformization</td><td>Adaptive Uniformization</td></tr>
<tr><td colspan="2">Read Q into sparse matrix structure
$\lambda = \max(-q(i,i)) \mid i \in S$, v = initial state</td></tr>
<tr><td>$P = I + Q/\lambda$
Calculate jump probabilities $U_{N_{sl}}(t)$,
$\ldots, U_{N_{sr}}(t)$ given t, λ and ϵ,
with $U_i(t) = 0$ for $i < N_{sl}$</td><td>$\lambda_0 = -q(v,v)$
Calculate $U_0(t)$ with ACE, Modified ACE
or LU using t, λ_0 and ϵ_B
$sum = U_0(t)$; $n = 1$; $done = 0$; $converged = 0$</td></tr>
<tr><td colspan="2">$\underline{\pi}_0(v) = 1$, $\underline{\pi}_0(i) = 0$ for $i \neq v, i \in S$
$\underline{\pi}(t) = \underline{\pi}_0 U_0(t)$</td></tr>
<tr><td>For $n = 1$ to N_{sr} {

$\underline{\pi}_n = \underline{\pi}_{n-1} P$</td><td>While $done = 0$ {
If $converged = 0$
$\underline{\pi}_n = \underline{\pi}_{n-1}(I + Q_{n-1}/\lambda_{n-1})$
Find λ_n given $A_n = \{i \in S \mid \underline{\pi}_n(i) > 0\}$
If $\lambda_n = \lambda$
$converged = 1$; $P = I + Q/\lambda$
LU: Find $U_n(t), \ldots, U_{N_a}(t)$
Modified ACE: convert to linear scheme
Else If LU Calculate $U_n(t)$
Else
$\underline{\pi}_n = \underline{\pi}_{n-1} P$
ACE or Modified ACE: Calculate $U_n(t)$</td></tr>
<tr><td colspan="2">$\underline{\pi}(t) = \underline{\pi}(t) + \underline{\pi}_n U_n(t)$</td></tr>
<tr><td>}</td><td>$sum = sum + U_n(t)$
If $(1.0 - sum) < \epsilon$
$done = 1$; $N_a = n$
$n = n + 1$
}</td></tr>
</table>

Figure 3.1: SU and AU Algorithms

Equation 3.20 as outlined in Section 3.1.3. Given the nature of the AU algorithm, it can be seen without much difficulty that Q could be generated as the transient solution was being found, making it applicable to unbounded state space problems. The implementation here depends on a pre-generated Q from the software package *UltraSAN* [2], and thus is limited to finite state space models.

Both algorithms use loops to control the summation of Equations 2.5 and 2.12. Each iteration calculates the next $\underline{\pi}_n$, then sums the product of it and epoch n's probability into $\underline{\pi}(t)$. From Figure 3.1, it is evident that this is a simpler process for SU, however, it is not necessarily less computationally intensive. In AU, finding $\underline{\pi}_n$ from $\underline{\pi}_{n-1}$ is dependent upon the adaptive uniformization rate λ_{n-1} and whether or not the rate has converged. If the rate is unconverged, $\underline{\pi}_n$ is calculated, and λ_n is found from the new set of active states. If, on the other hand, the converged rate λ has already been reached, $\underline{\pi}_n$ is found as in SU. In this implementation, as soon as the maximum, or converged, rate in the infinitesimal generator matrix is reached, Q is permanently converted to P so that no repetitive computation is done.

The other main difference in the two methods is in how the main loop terminates. In SU, the end N_{sr} is determined *a priori* from the Fox and Glynn algorithm [5]. On the other hand, this implementation of AU finds the λ_i's on the fly, and thus jump probabilities are calculated only an epoch at a time. These are summed iteration by iteration and checked at every update to see if the ϵ requirement has been met. Thus N_a is not determined until the end of the computation. More details regarding the implementations can be found in Appendix D.

CHAPTER 4

PROGRAM INSTRUMENTATION

In order to fairly assess the accuracies and computational complexities of the different algorithms, a *C++* class was developed with which the programs were instrumented. First, it enables the counting of floating point arithmetic operations by both operation type and program phase. And second, it tracks the bound on the machine round-off error for all floating point data. Thus this class is very useful for comparing numerical algorithms in terms of complexity and accuracy. Through its use of operator overloading [30], this extended double class easily instruments numerical *C* or *C++* code, requiring only changes to the variable type declarations and output statements.

Associated with each class element are pointers to counters for addition, subtraction, multiplication and division operations. These counters are double floating point types in order to increase the count range by several orders of magnitude over unsigned long integers. Every floating point operation involving an extended double type increments the appropriate counter pointed at by the data element. For example, the statement '`a = b + (c / d)`' increments the addition and division counters pointed to by `a`, as well as updating the round-off error bound on `a`. By assigning different counters to variables in different segments of the program, computational complexity can be broken down into finer algorithmic detail.

The computed data $\tilde{x}$ and round-off error bound b are stored as double floating point types in each element of the class, such that the true x must lie in the range $\tilde{x} \pm b$. The bound b is not related to approximations due to the algorithm in use, but is only a bound on the machine round-off error based on the previous computations necessary to find $\tilde{x}$. Initial round-off error is determined by the machine precision as a default, although external values can be assigned. For a given arithmetic operation, b is propagated as follows

$$\text{For } \tilde{x}_r = \tilde{x}_1 + \tilde{x}_2,\ b_r = b_1 + b_2 \tag{4.1}$$

$$\text{For } \tilde{x}_r = \tilde{x}_1 - \tilde{x}_2,\ b_r = b_1 + b_2 \tag{4.2}$$

$$\text{For } \tilde{x}_r = \tilde{x}_1 \times \tilde{x}_2,\ b_r = b_1 \text{abs}(\tilde{x}_2) + b_2 \text{abs}(\tilde{x}_1) + b_1 b_2 \tag{4.3}$$

$$\begin{aligned} &\text{For } \tilde{x}_r = \tilde{x}_1 \div \tilde{x}_2, \\ &b_r = \begin{cases} \max\left\{\text{abs}\left(\frac{\tilde{x}_1}{\tilde{x}_2} - \frac{(\tilde{x}_1 - b_1)}{(\tilde{x}_2 + b_2)}\right), \text{abs}\left(\frac{(\tilde{x}_1 + b_1)}{(\tilde{x}_2 - b_2)} - \frac{\tilde{x}_1}{\tilde{x}_2}\right)\right\} & \text{if } \tilde{x}_1 \times \tilde{x}_2 \geq 0 \\ \max\left\{\text{abs}\left(\frac{\tilde{x}_1}{\tilde{x}_2} - \frac{(\tilde{x}_1 + b_1)}{(\tilde{x}_2 + b_2)}\right), \text{abs}\left(\frac{(\tilde{x}_1 - b_1)}{(\tilde{x}_2 - b_2)} - \frac{\tilde{x}_1}{\tilde{x}_2}\right)\right\} & \text{if } \tilde{x}_1 \times \tilde{x}_2 < 0 \end{cases} \end{aligned} \tag{4.4}$$

where $\text{abs}(\tilde{x})$ is the absolute value of $\tilde{x}$ and $\max\{\}$ returns the maximum of its arguments. This method provides an upper bound [29] estimate on the error accumulated due to machine round-off, but cannot be considered completely rigorous because the bound computations themselves are subject to round-off error. In the next chapter results based upon this class will be given.

CHAPTER 5

RESULTS

In this chapter, three models are discussed and used for comparison of the methods. First, these models are described in the context of SAN modeling and their AU attributes, followed by a listing of the goals of this chapter. These include comparison of each method's stability, an illustration of the operation counting *C++* class, and finally, an evaluation of AU and SU's performance.

5.1 Example Markov Models

To study the viability of AU, and to illustrate the code instrumentation, three example models are used. The first is an extended machine-repairman (EMR) model with repair delayed. It is a kind of reliability model particularly well-suited for solution by AU, and can easily be extended in terms of state-space size and its jump process by varying parameters. The last two are reliability models of real systems with repair. One is a fault-tolerant parallel computer system with several layers of redundancy while the other is a distributed database architecture. These systems and their associated parameters will be explained in greater detail in the following sections, while the formal documentation for their *UltraSAN* models can be found in the appendices. A final section tabularizes the resulting adaptive uniformization rates of the models.

5.1.1 EMR Model

The EMR model is similar to ones given in [3] and [31]. In this model there are K components, all with the same failure rate, ρ. There are two classes of failure, *hard* and *soft*, where the probability of a failure being soft is c and that of a hard failure $1 - c$. c is known as the *coverage factor*. Repair begins when only r components remain functional, with repairs taking place at a rate of μ for a hard failure and ν for a soft failure, with $\nu > \mu$ typically. The failures and repairs are all negative exponentially distributed in time with ρ, μ and ν as parameters. Each component has independent repair capability, and repair continues until all units are operational. Correct service is provided whenever at least one component is functional. Usually μ, $\nu \gg \rho$, and thus there are a number of epochs with low adaptive uniformization rates followed by higher rates and the converged rate.

For this study, two sets of models with a range of K values and constant r, ρ, μ, ν and c have been chosen. Notice that c has no effect on the state space size, assuming c is in $(0, 1)$, and that the adaptive uniformization rates are a function of K, r, ρ and $\max\{\mu, \nu\}$. Also, the converged rate $\lambda = (K - 1)\max\{\mu, \nu\} + \rho$. For the first set $K = 20$ or 50, $r = 10$, $\rho = 1$, $\nu = 1000$, $\mu = 800$ and $c = 0.5$. Thus while μ and c may effect the reliability, they do not effect the computation time in the discussed algorithms. The second set consists of larger models, where $K = 200$, 250 or 300, $r = 100$, $\rho = 1$, $\nu = 100$,

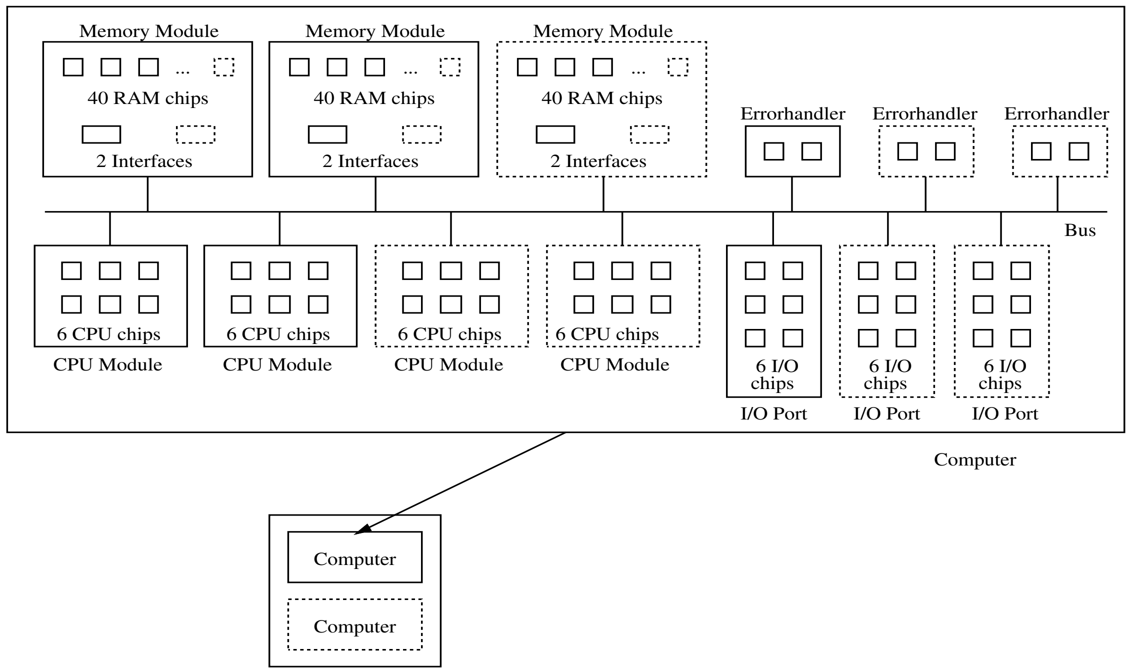

Figure 5.1: Fault-Tolerant Parallel Computing System

$\mu = 80$ and $c = 0.5$. For all LU results, $\epsilon_B = 10^{-14}$, whereas parameter ϵ is varied along with t as part of the experiment. Mission times t on the order of $1/\rho$ were used, the mean time of a component failure, which is reasonable for highly reliable applications [18]. A typical measure of interest in this model is then system reliability, or the probability of at least one unit functioning.

5.1.2 Fault-Tolerant Parallel Computer System Model

A more realistic test-model is a fault-tolerant multiprocessor system (FTPCS) originally given in [12], and modeled with *UltraSAN* in [26]. The version here has been modified slightly to include delayed repair and greater redundancy, and is shown in Figure 5.1. The system consists of two computers, each composed of four CPU units, two of which are spare, three memory modules, one of which is spare, three I/O ports with two redundant, and three error-handlers, two of which are spare. In addition, each memory module has 40 RAM chips, one of which is spare, and two sets of interface chips with one set spare. Repair only takes place when an entire computer has failed, with total system failure occuring when both computers have failed. In Figure 5.1, all spare units are indicated with dashed lines.

Two varieties of the model were used in this study, one with perfect coverage, i.e., spares cover failures 100% of the time, and the other with non-perfect coverage. The coverage factors are given in Table 5.1, and are an extension of those from [12]. Finally, the chip failure rate is assumed to be 100 failures per billion hours, as in [12], with a negative exponential distribution. Parts with multiple chips have failure rates in proportion to their chip count. Repair is also assumed to occur with an exponential distribution, with a mean of 0.001 years. Mission times of interest are on the order of years.

Redundant Component	*Fault Coverage Probability*
RAM Chip	0.998
Memory Interface	0.998
Memory Module	0.95
CPU Unit	0.995
I/O Port	0.99
Error-handler	0.99
Computer	0.95

Table 5.1: Coverage Factors for FTPCS

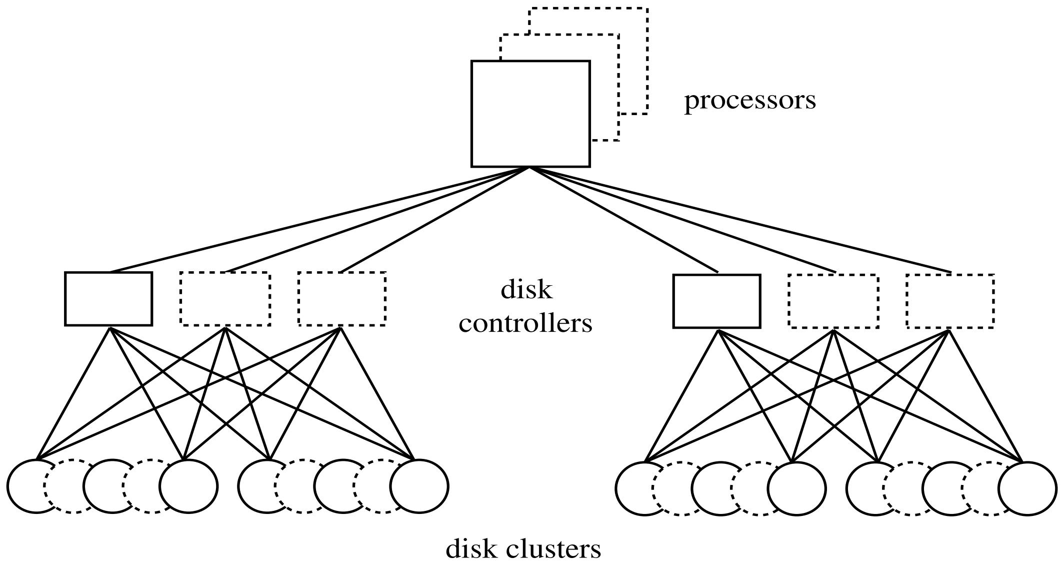

Figure 5.2: Distributed Database System

5.1.3 Distributed Database Model

The last example is a distributed architecture for a database system (DDA), based on a model from [26], which was itself modified from [21, 22]. The system consists of four disk clusters, two sets of controllers, and three processors as depicted in Figure 5.2. A disk cluster has five disks, with information redundant between them such that any two can fail without loss of data access. A disk controller set consists of three controllers, at least one of which must be functioning. The system is then operational when all clusters are fully accessible through at least one controller by at least one of the three processors, and coverage factors are not considered here. Repair for the different components is independent. Repair on a disk cluster begins when two disks have failed, and controllers and processors are repaired when two of three in a set have failed. All redundant components are shown with dashed outlines in Figure 5.2. Table 5.2 gives failure and repair rates per hour for the database system components. Mission times of interest are on the order of weeks.

Component	*Failure Rate*	*Repair Rate*
Processor	1/2000	1
Disk Controller	1/2000	1
Disk	1/6000	1

Table 5.2: Per Hour Failure and Repair Rates

5.1.4 Adaptive Uniformization Rates

Since the adaptive uniformization rates form the jump process, they play a key role in determining how beneficial AU is. Table 5.3 lists the rates for each model, λ_0 through $\lambda_m = \lambda$, along with m values. Therefore, the last rate in each column is the maximum and converged rate for each system.

5.2 Goals of Results

Several aspects of the results will be discussed. First, the accuracy of the different methods will be compared in terms of their resulting bounds on the round-off error. Second, an illustration of operation counting in the instrumented code will be given. Finally, the computational complexity of the algorithms are compared. Two methods of comparison are used, computation time (CPU usage) and number of floating point operations. Since all AU algorithms have a higher order of complexity in computing jump probabilities than SU with respect to time, the results exhibit *crossover points* t', where AU is less computationally complex for $t < t'$ and SU performs better for $t > t'$,

EMR					*FTPCS*		*DDA*
$K = 20$ $r = 10$	$K = 50$ $r = 10$	$K = 200$ $r = 100$	$K = 250$ $r = 100$	$K = 300$ $r = 100$	perfect coverage	non-perfect coverage	
20	50	200	250	300	0.1613	0.1613	0.0078
19	49	199	254	299	0.1604	1000.0806	0.0077
18	48	198	253	298	0.1613		1.0075
17	$\vdots$	$\vdots$	$\vdots$	$\vdots$	1000.0806		1.0077
16	12	102	102	102			2.0072
15	11	101	101	101			2.0073
14	40010	10100	15100	20100			3.0068
13	41009	10199	15199	20199			3.0070
12	$\vdots$	$\vdots$	$\vdots$	$\vdots$			4.0065
11	49001	19901	24901	29901			4.0067
10010							5.0055
11009							5.0060
12008							6.0045
13007							6.0050
14006							7.0035
15005							7.0040
16004							7.0045
17003							7.0052
18002							7.0053
19001							7.0055
							7.0057
$m = 19$	$m = 49$	$m = 199$	$m = 249$	$m = 299$	$m = 3$	$m = 1$	$m = 20$

Table 5.3: Adaptive Uniformization Rates of the Example Models

whenever $N_a < N_{sr}$. The crossover point based on computation time is designated t_T, while the floating point operation crossover time is labeled t_F. The two measurements will show slight differences because some overhead is not accounted for by operation counts, and the operation type distribution varies between SU and AU.

5.3 Comparison of Method Stability

Both SU and LU are numerically stable owing to their use of probability matrices with positive, bounded elements, and the Fox and Glynn method to calculate Poisson probabilities, while ACE and Modified ACE are not guaranteed to be stable. The limitations of these algorithms will be looked at first through the use of the *C++* class. The round-off error results are from system reliability, and are normalized. By this it is meant that for a given time t, the round-off bound $B_r(t)$ of system reliability $R(t)$ is normalized to $B_r(t)/R(t)$. Thus the round-off errors shown are relative to 1.0, i.e.

$$1.0 - \frac{B_r(t)}{R(t)} \leq \frac{R(t)}{R(t)} \equiv 1.0 \leq 1.0 + \frac{B_r(t)}{R(t)}, \tag{5.1}$$

where $B_r(t)/R(t)$ is the actual data used in all stability plots. This enables fair comparison over a range of t and $R(t)$ values. As can be seen in Figure 5.3, the accuracy of both ACE and Modified ACE is limited. The tests for Figure 5.3 were run on the $K = 20$ EMR model with the requirement $\epsilon = 10^{-4}$, and yet the bounds on the resulting round-off error were often orders of magnitude larger than this for a large range of t. Other EMR models with different K values, and smaller ϵ requirements, showed worse performance, even to the point where any viable solution was unattainable.

In order to get a better picture of ACE and Modified ACE stability, data has been collected from the other two models as well. In Figure 5.4, plots with differing time points are shown in terms of error bound versus desired accuracy, ϵ, for the Distributed Database Architecture model. Performance degrades rapidly both in terms of increasing time and decreasing ϵ for AU with ACE, as both of these factors cause an increase in the necessary

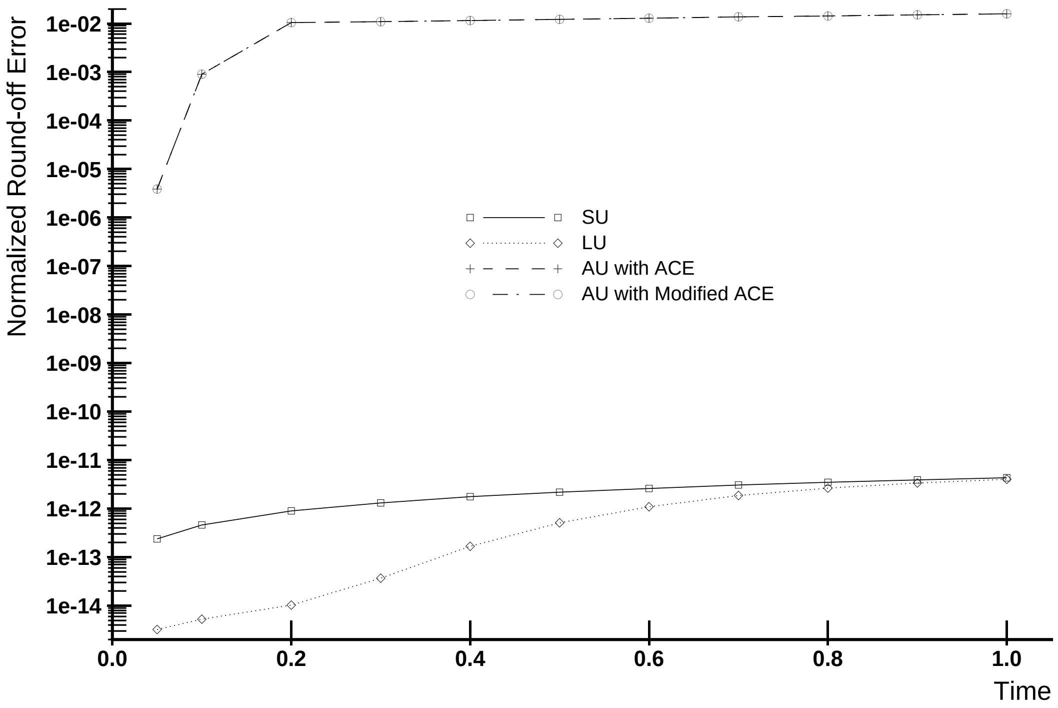

Figure 5.3: Reliability Round-off Error Bounds: $K = 20$, $r = 10$, $\epsilon = 10^{-4}$

truncation point and number of jump probabilities to be calculated. In fact, the missing points in the AU with ACE plots have round-off errors outside the scale of the graph. Modified ACE recorded the exact same results as ACE because the converged rate was never reached.

Of all models used, FTPCS with perfect coverage was best suited to solution via AU with ACE or Modified ACE, as depicted in Figure 5.5, where once again, round-off error is normalized by system reliability. In particular, Modified ACE shows good stability for a large range of t and fairly small ϵ. Further study, however, shows that even for this model, ACE and Modified ACE methods are limited compared to LU. Figure 5.6 is a graph of truncation point, or number of iterations until termination, versus decreasing

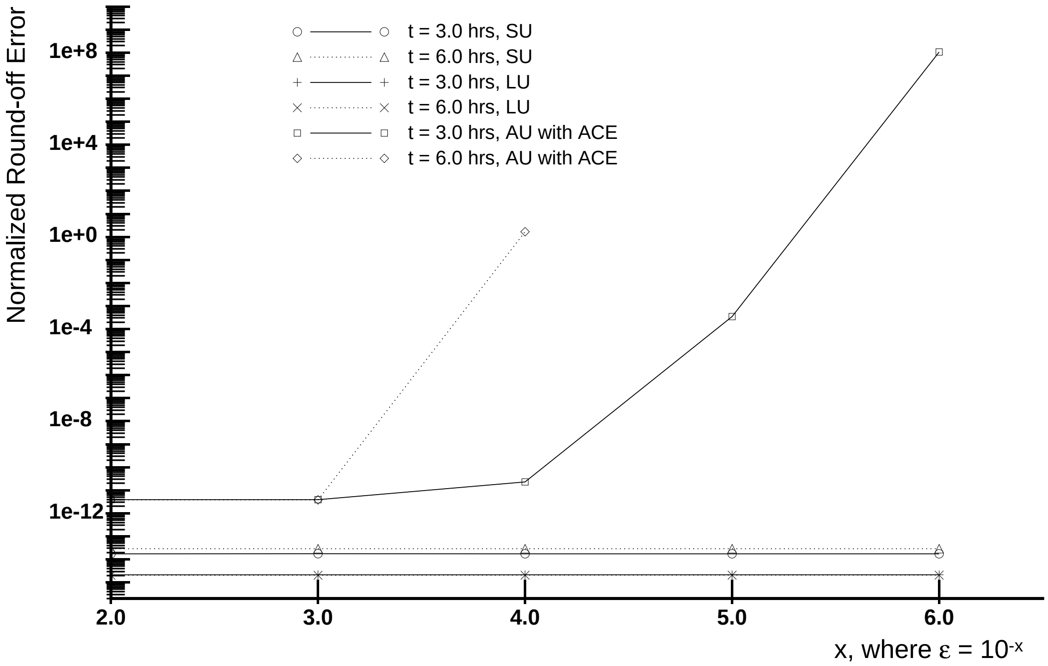

Figure 5.4: DDA Model: Round-off Error vs. ϵ

ϵ. Termination occurs when either accuracy demands are met or instability detected. Instability is said to have occurred when either the computed jump probabilities fall outside [0,1], or the summation of all these probabilities comes to greater than 1. It can be seen that as ϵ drops below 10^{-6}, the curves for ACE and Modified ACE incorrectly flatten out to a slope of zero, indicating their inability to solve to very high accuracies because numerical instabilities cause premature termination. LU does not suffer from this drawback. Thus, due to the limited and unpredictable capabilities of the ACE and Modified ACE algorithms, they are probably not useful for general implementations of adaptive uniformization. Therefore, the remaining performance results focus on the LU variant of AU.

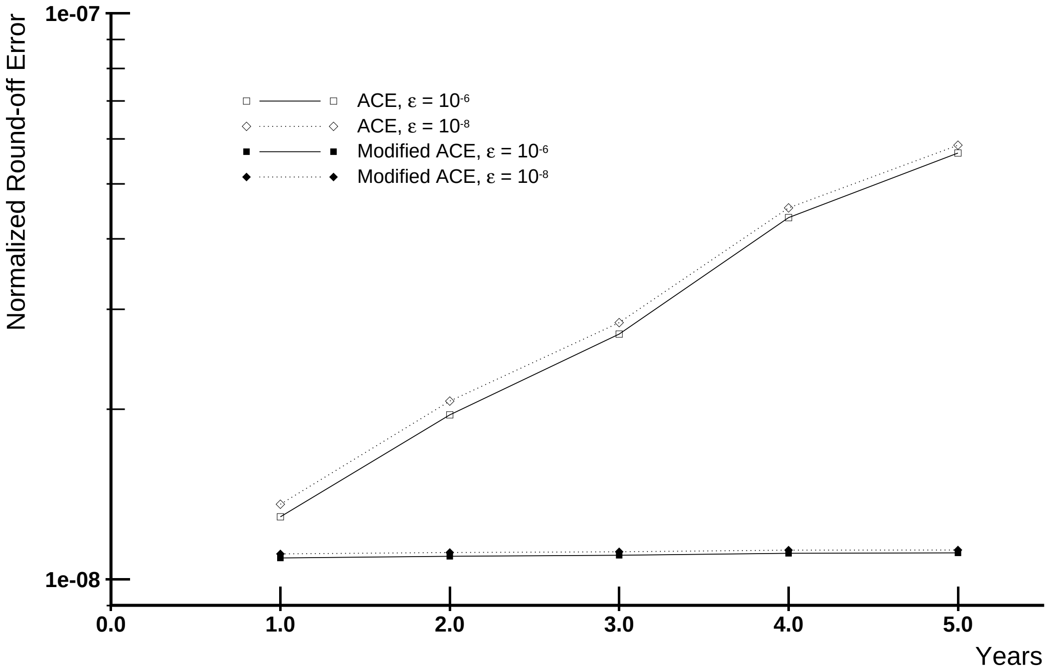

Figure 5.5: FTPCS Model: Round-off Error vs. Time

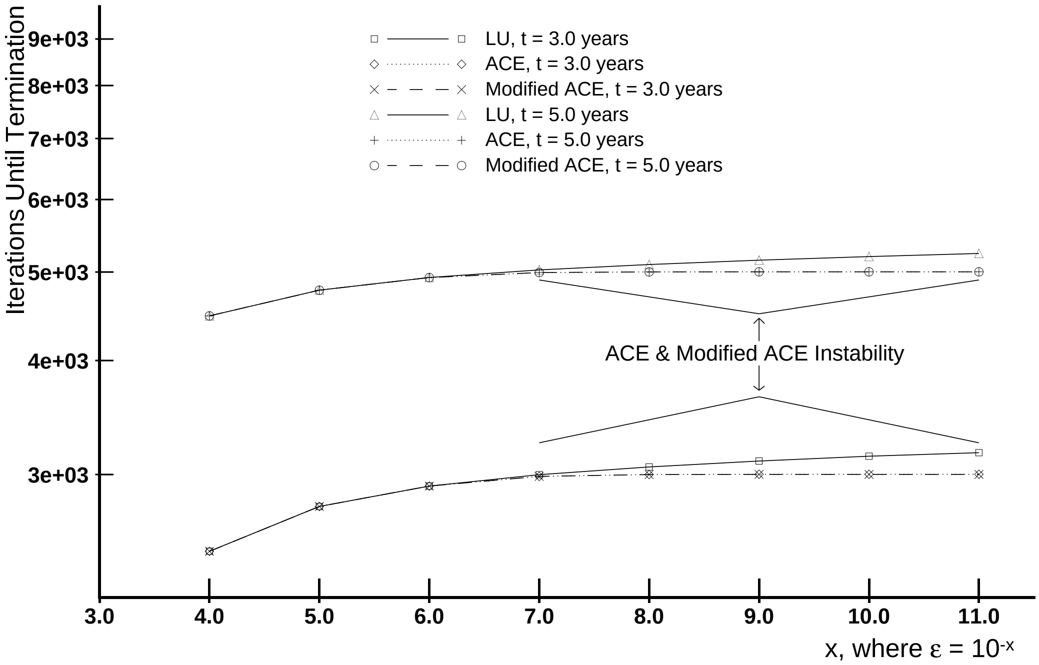

Figure 5.6: FTPCS Model: Iterations vs. ϵ

On the other hand, SU and LU show good results in terms of numerical stability, with similar results found over a wide range of models and ϵ values as demonstrated in Figures 5.3 and 5.4. LU shows better performance than SU in Figure 5.3 for small t, and in Figure 5.4, simply because fewer calculations were necessary to achieve the result. Note that the truncation error in SU can be made arbitrarily close to 0 through the use of Fox and Glynn's algorithm for computing Poisson probabilities. However, making ϵ less than the machine's epsilon, i.e., the smallest number x such that $1.0 + x \neq 1.0$, has little value. LU's truncation error is limited by this machine epsilon, since a running total sum of the jump probabilities is used to detect when to truncate. A specified ϵ less than the machine epsilon would not allow the infinite sum to truncate.

5.4 Illustration of Operation Counting

This section shows results to illustrate use of the instrumentation method to count operations. Figure 5.7 shows a breakdown by operation type for a 50 component EMR model. The operation types are important because their individual computational complexities vary from machine to machine, and thus comparisons may want to take this into account. In this example it can be seen that additions and multiplications dominate the divisions and subtractions in both the SU and LU algorithms. However, the operation distributions are not the same, as LU requires a larger percentage of subtractions and divisions for solution. The operation crossover point in computational complexity can be seen to lie somewhere between $t = .6$ and $t = .8$.

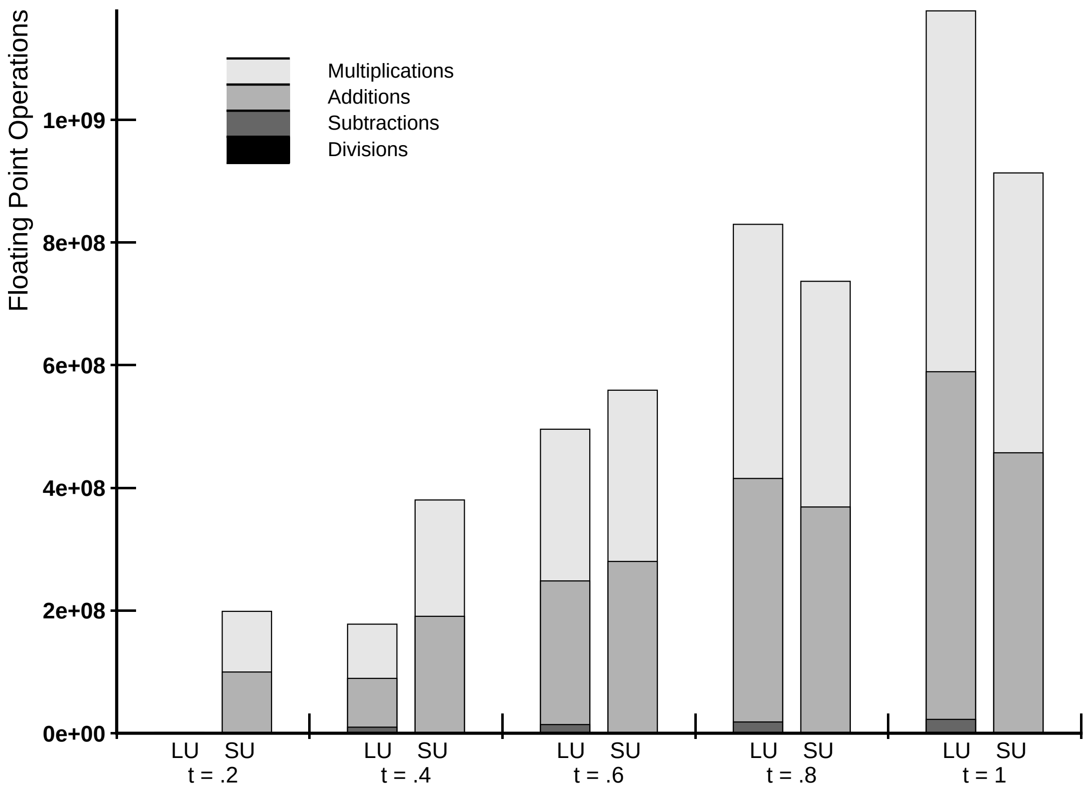

Figure 5.7: $K = 50$, $r = 10$, 2095 states, $\epsilon = 10^{-12}$

In Figure 5.8 the counts are divided between the different aspects of the algorithm that contribute to the computational complexity:

- INIT: Operations done during the read-in of Q.
- JPR: Computation of jump probabilities $U_n(t)$.
- QP: Calculation of P and/or P_n.
- MVM: Matrix vector multiplication $\underline{\pi}_{n-1}P$ and/or $\underline{\pi}_{n-1}P_{n-1}$.
- SOL: $\underline{\pi}(t) + \underline{\pi}_n U_n(t)$ calculation.

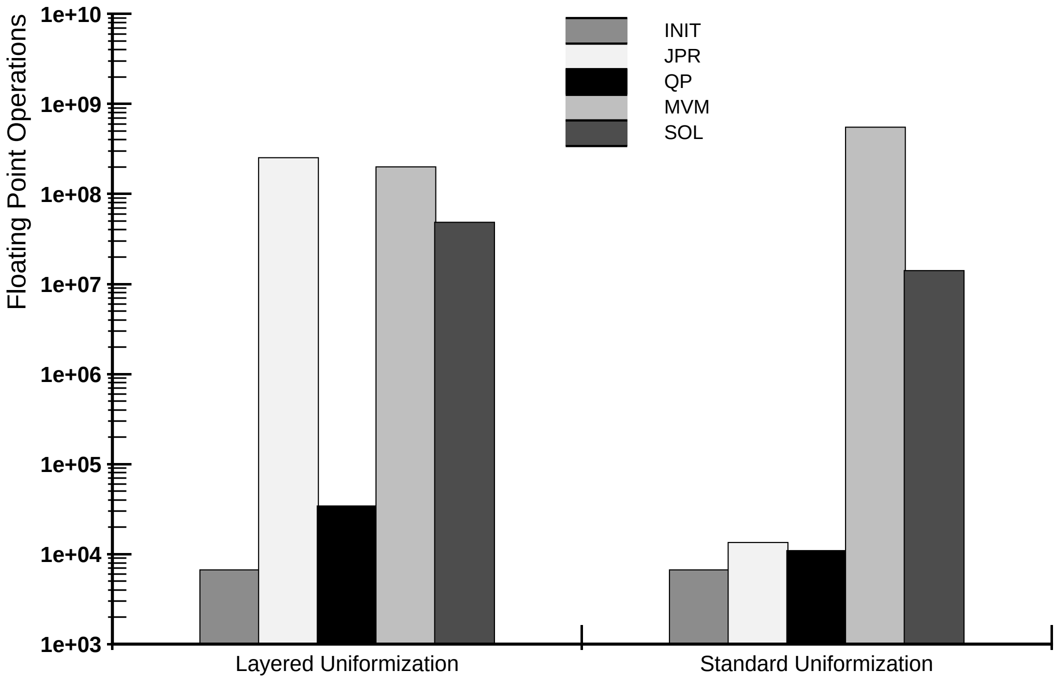

Figure 5.8: $K = 50$, $r = 10$, 2095 states, $\epsilon = 10^{-12}$, $t = .6$

The time $t = .6$ was selected because total computational complexity is almost equal at this time, allowing fair comparison. The histogram clearly shows the trade-off between the two methods. LU trades increased computational complexity in calculating the jump probabilities (JPR) for a reduction in the number of matrix vector multiplications (MVM), as compared to SU. The latter follows from the fact that LU truncates earlier because the adaptive uniformization rates allow it to make probabilistically larger jumps. Note that since the scale is logarithmic, the difference in MVM complexity is actually greater than the difference in JPR. LU also shows increased activity in QP since P_n is calculated multiple times and in SOL because LU does not left truncate.

5.5 Comparison of AU and SU

The next two figures demonstrate the computational advantages of LU, as well as illustrate the two kinds of crossover points using the EMR model. Larger state space-sized models with $K = 200$, 250, 300 and $r = 100$ were selected to show AU and SU's ability to solve very large problems. Figure 5.9 depicts the computational complexity in terms of the total number of floating point operations. Within these results, all operation types are given equal value. For a given machine these operations could easily be weighted to reflect their individual computational intensities. It can be seen that t_F, the operation crossover point, occurs at approximately $t = 1.9$. For $t < .5$, the converged rate has not been reached when the truncation point occurs, and thus it can be seen that savings of about 2 orders of magnitude are reaped for this particular model. The large jump in complexity at $t \approx .5$ occurs when the converged rate is reached, and no further savings are possible.

A graph of computation time on a Sparcstation 10/30 versus time t is given in Figure 5.10. The t_T occurs at slightly less than $t = 2$. This shows that the extra complexity in the LU algorithm has little to no effect on the overall computation time. Also note that the savings for $t < .5$ are less in terms of time because operation counts do not reflect computational time spent on matrix read-in, etc. If, for example, the matrix were passed directly from the CTMC generator, this would not be a factor. The rest of the results will be based upon instruction counts, since these are reliable, and repeatable across hardware and software platforms. CPU usage is not the most dependable measure since

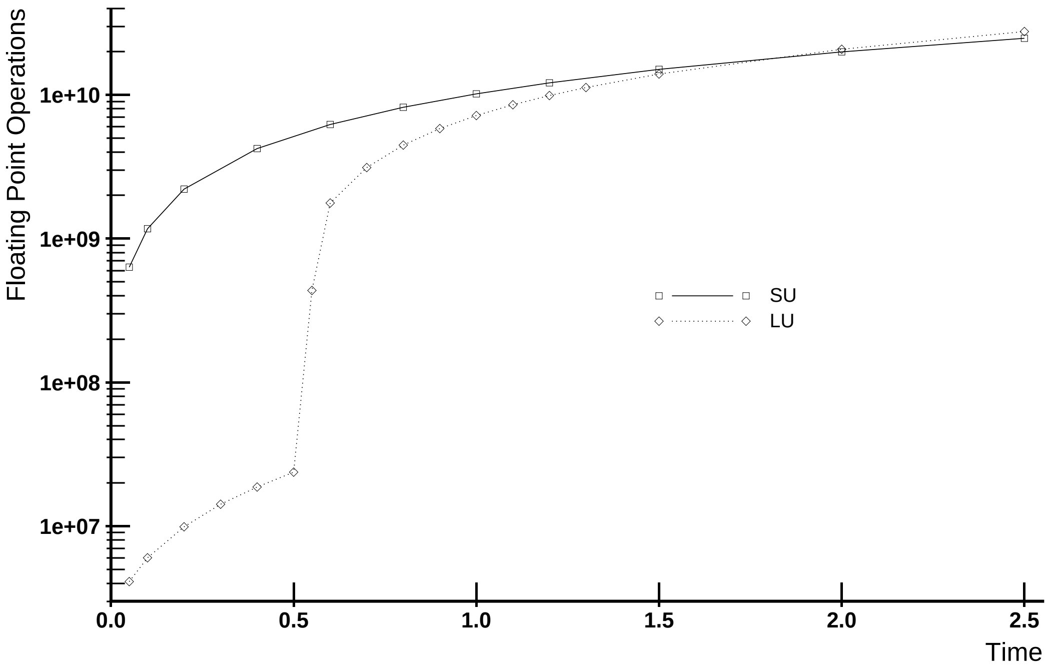

Figure 5.9: Operations vs. Time: $K = 250$, $r = 100$, 42700 states, $\epsilon = 10^{-8}$

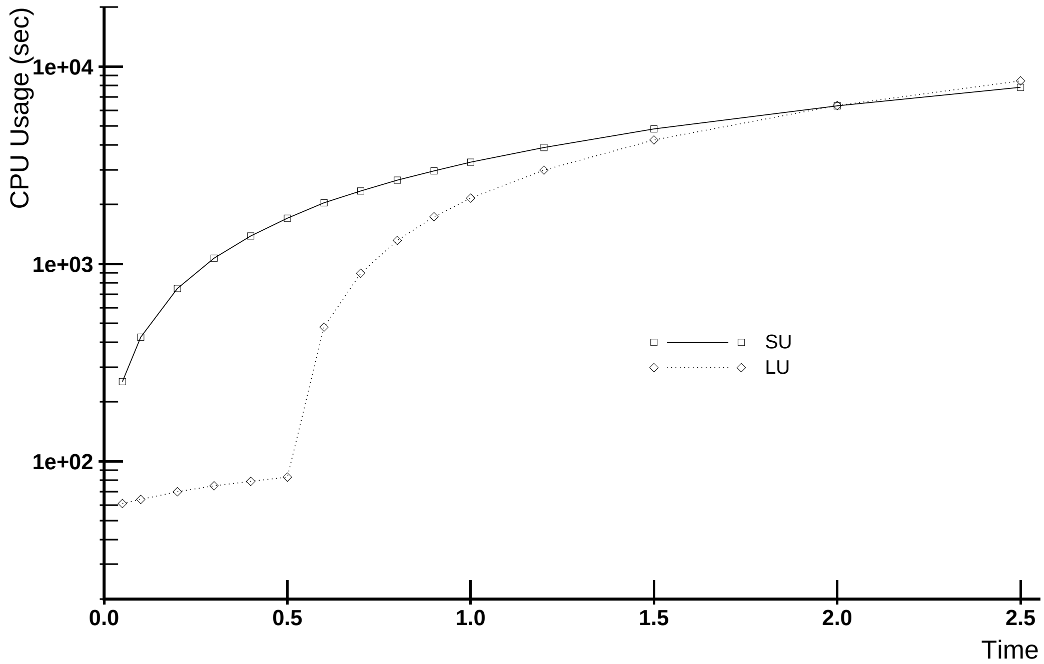

Figure 5.10: CPU Usage vs. Time: $K = 250$, $r = 100$, 42700 states, $\epsilon = 10^{-8}$

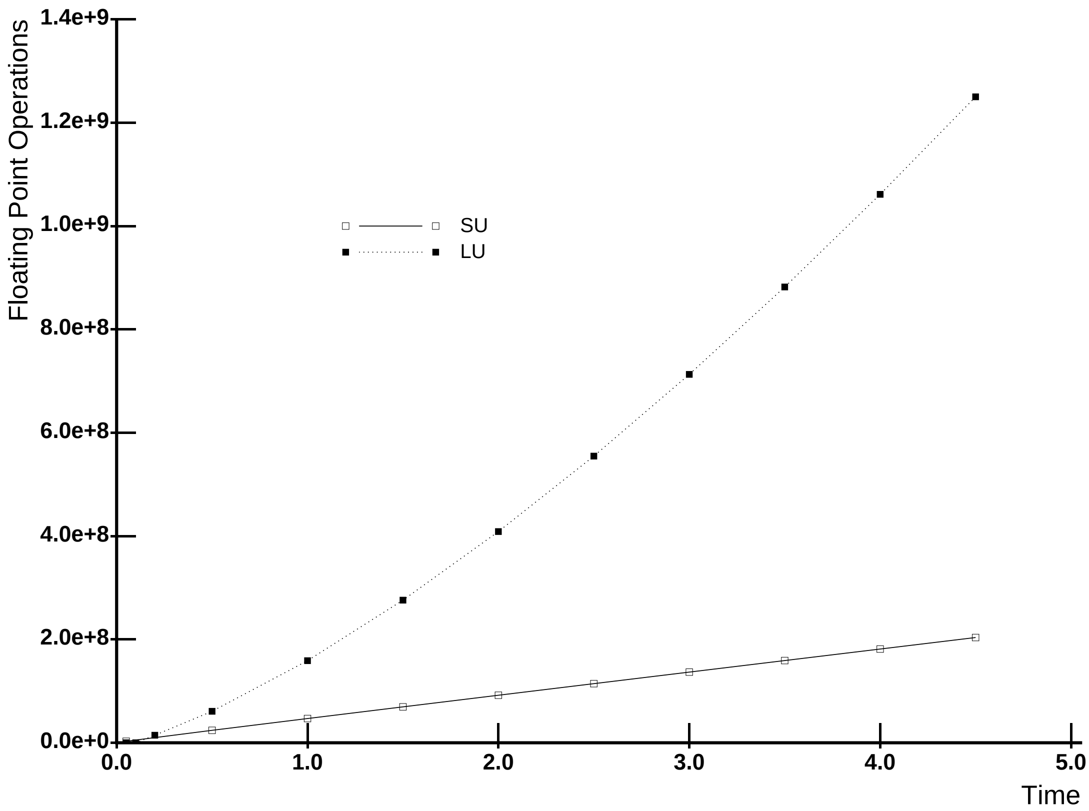

Figure 5.11: Complexity vs. Time: $K = 20$, $r = 10$, 265 states, $\epsilon = 10^{-4}$

it can be unpredictably influenced by concurrent execution of the operating system and other application programs.

In order to see that the two algorithms, LU and SU, have different orders of complexity in terms of big-O notation, Figure 5.11 is given. This EMR model is not particularly amenable to LU solution because of its small state-space size and lesser number of unconverged rates. However, these same factors enable solution over a large time scale, allowing general complexity trends to be seen. It is evident that SU is linear with respect to time, while LU is of higher order. In fact, it is the jump process calculation with complexity

$O(N_a\sqrt{N_{Br}})$ that dominates and gives LU its upwardly curving shape. Still, there exists a crossover point t_F, at approximately $t = .15$.

Figure 5.12 gives results for the two FTPCS models. The model of the system with perfect coverage has 23653 states, while the non-perfect coverage model has 23869 states and a denser transition matrix. From the graph, the former has a crossover point between 6 and 7 years, while the latter's t_F is almost at 3 years. The model with perfect coverage does better since the jump process has more unconverged rates (see Table 5.3). Non-perfect coverage removes the delayed repair aspect from the solution's point of view. The effect of this is more evident in Figure 5.13. The model with non-perfect coverage never shows truly substantial savings because it only has a single unconverged rate, whereas for perfect coverage the time of interest t must be greater than 0.1 years for the converged rate to be reached. The flatness for the LU graph of FTPCS with perfect coverage, for $t \leq 0.1$, is attributable to its small number of unconverged rates, making further savings minimal.

The third model, the distributed database architecture, is used for comparison in Figure 5.14. It is an 8260-state model, with a crossover point t_F at a little less than 1 week. Figure 5.15 compares SU and LU for smaller mission times. The point at which covergence occurs is not very clear in Figure 5.15 since many of the unconverged rates have values close to the converged one. Finally, notice in Figures 5.12 through 5.15 that LU does not have the orders of magnitude savings it has for EMR models. This is because with their small number of unconverged rates and lower converged rate, N_a and N_{sr} differ by less. However, while these models do not show as impressive of savings, they do show

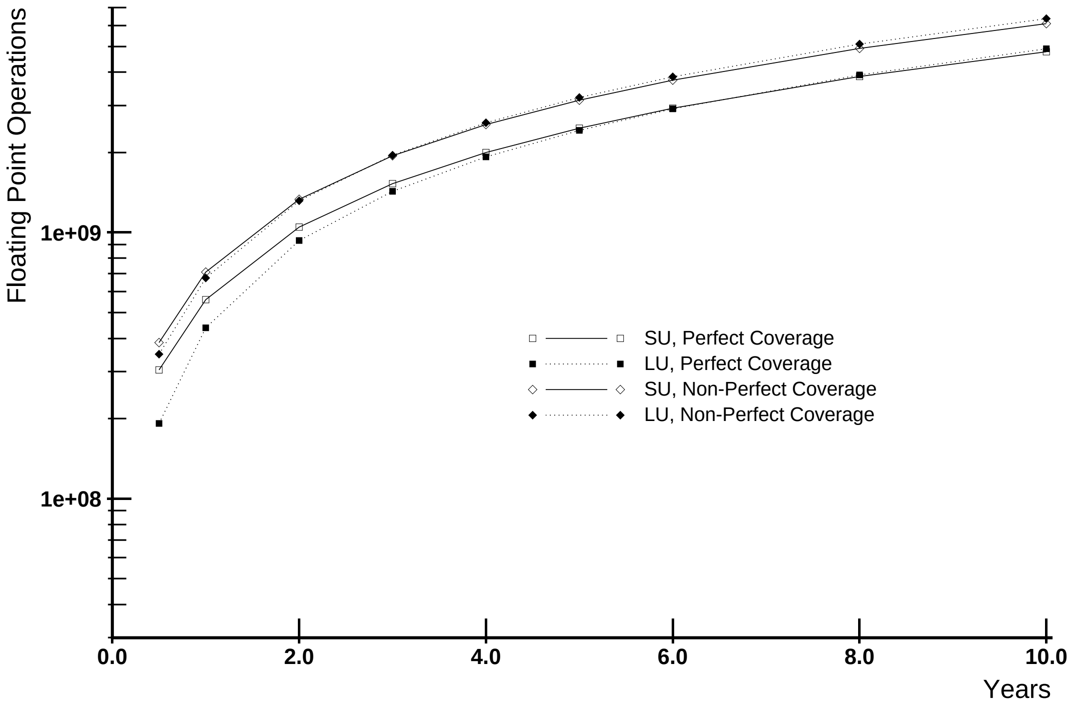

Figure 5.12: Complexity vs. Time: FTPCS Model, $\epsilon = 10^{-6}$

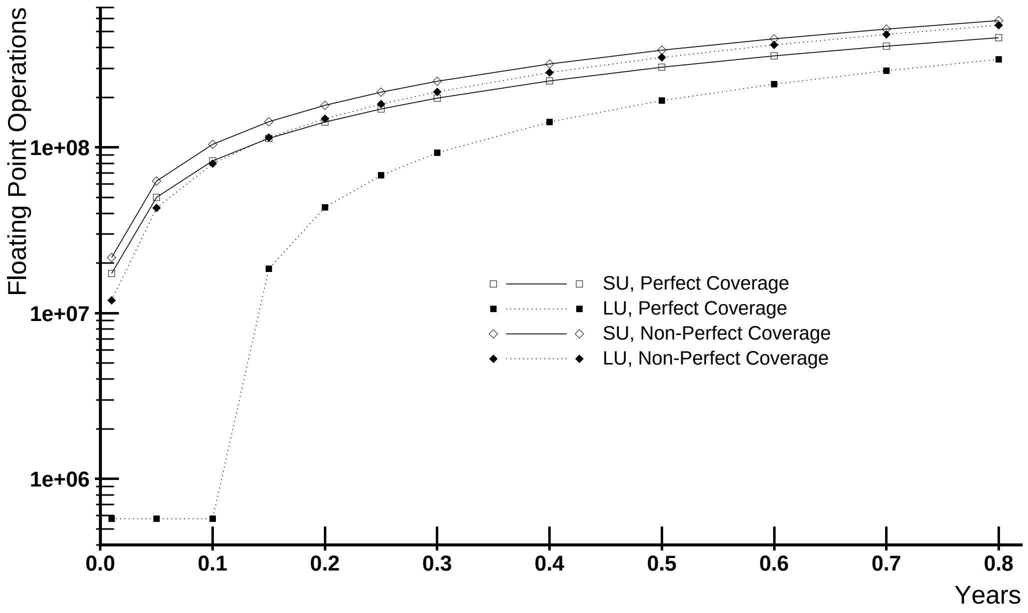

Figure 5.13: Complexity vs. Time: FTPCS Model, $\epsilon = 10^{-6}$

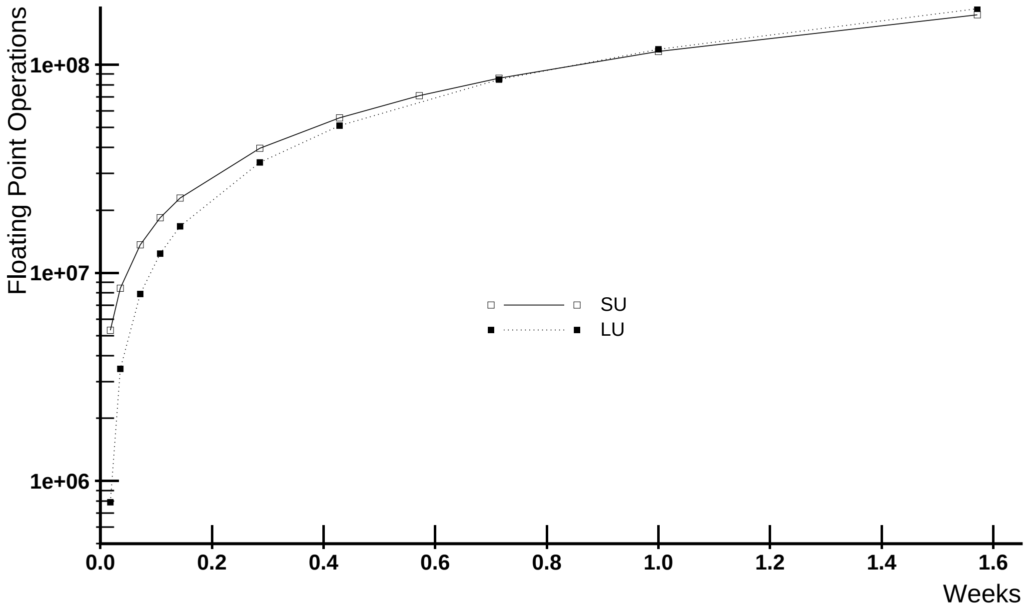

Figure 5.14: Complexity vs. Time: DDA Model, $\epsilon = 10^{-6}$

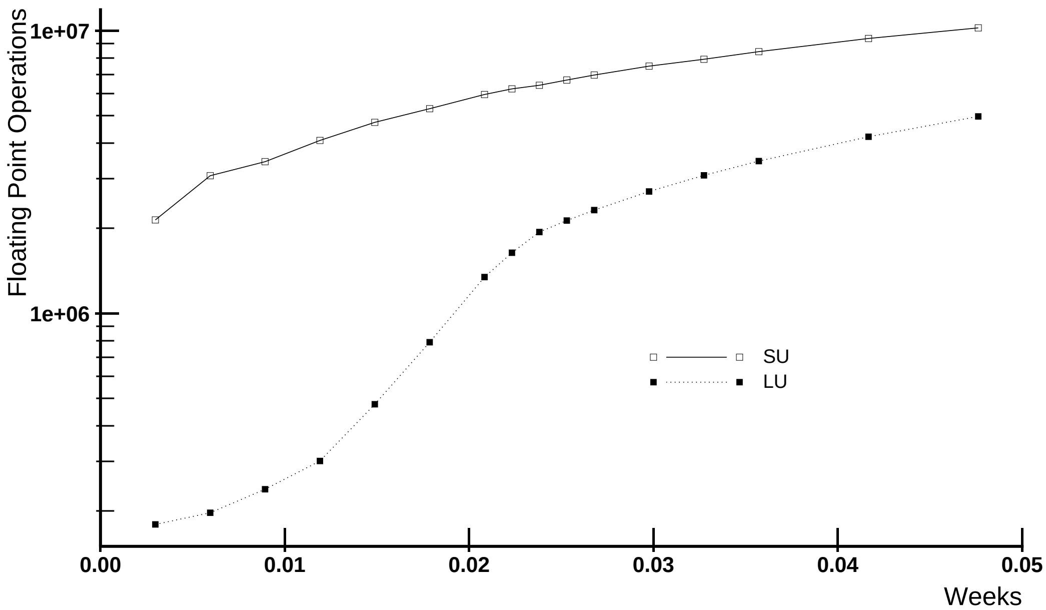

Figure 5.15: Complexity vs. Time: DDA Model, $\epsilon = 10^{-6}$

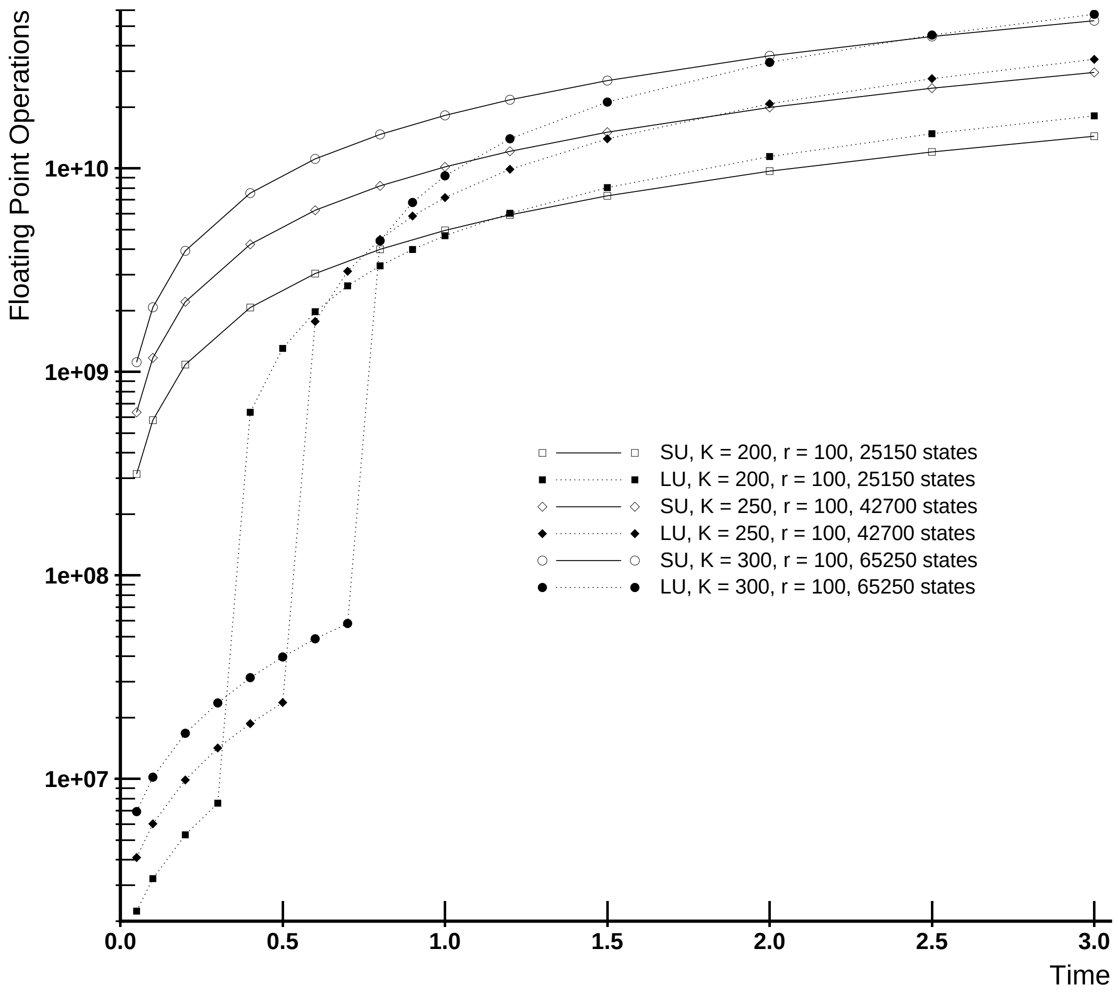

Figure 5.16: Complexity vs. Time: EMR Models with $\epsilon = 10^{-8}$

larger ranges of t for which savings exist. Also, further redundancy, perhaps necessitated by greater reliability needs, would improve the profitability of LU.

The computational complexity versus the time parameter for three different EMR models of varying state space size is shown in Figure 5.16. This graph depicts how increased state space size increases t_F because a reduction in the number of vector matrix multiplications becomes a more important factor. t_F rises from less than 1.2, to approximately

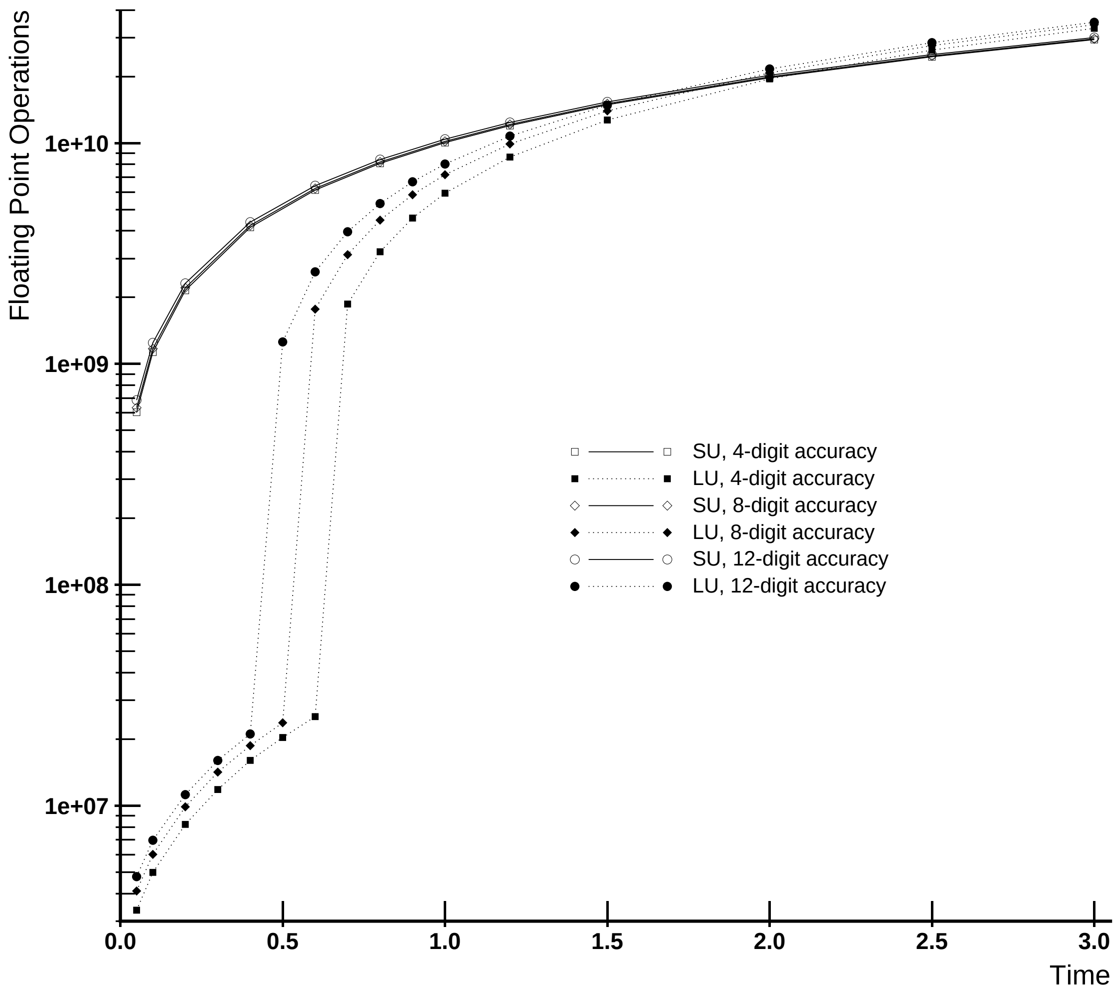

Figure 5.17: Complexity vs. Time: $K = 250$, $r = 100$, $\epsilon = 10^{-4}, 10^{-8}, 10^{-12}$

1.8, and finally to just under 2.5 for the 65250-state model. Note that this effect on t_F is reduced to some extent since as the state space size scales, so does its maximum, or converged rate, which has a greater effect on LU jump probability calculation than for SU.

Figure 5.17 illustrates how the requested truncation error limit effects the computation. Obviously, SU is insensitive to accuracy demands. This is because the Poisson distribution approaches zero rapidly in either tail, making the amount of truncation differ little with changes in ϵ. The distribution of the jump probabilities decreases much more slowly from its maximum in LU due to the jump process' higher variance, and therefore changes in ϵ significantly effect the truncation point N_a and thus the computational effort needed for solution. Therefore, it is seen that for $\epsilon = 10^{-4}$, $t_F \approx 2.1$, while for $\epsilon = 10^{-8}$, t_F is about 1.9 and for $\epsilon = 10^{-12}$, $t_F \approx 1.7$.

CHAPTER 6

CONCLUSIONS

Many methods have been proposed to compute the transient state occupancy probabilities in continuous-time Markov chains, but no method has proven best for all models and time points of interest. Uniformization is popular, but results in extremely long run times to achieve acceptable accuracy when models are stiff. Adaptive uniformization has been proposed to alleviate these problems for some models. In this thesis, the efficiency and stability of three implementations of AU were compared to that of standard uniformization. The efficiency and stability is compared by instrumenting the code using a *C++* class. This class was developed to track the bound on the round-off error incurred during the computation, as well as its computational complexity, measured in number of floating point operations. Operation counts can be refined by operation types and the algorithm phase in which they occur. The class is general, and can instrument other numerical algorithms written in *C* or *C++* code with minimal effort.

The three adaptive uniformization implementations studied differ in the method by which they compute the jump probabilities. Of the three methods, ACE has the highest order of complexity, but is theoretically applicable to any jump process resulting from AU. Modified ACE has the least complexity, but can only be used when and if a converged rate exists. Standard uniformization is the third method, hence the name layered uniformization for this AU variant. LU's order of complexity falls between the two previous methods, and because of its probabilistic nature, is numerically stable. Comparing the AU alternatives with SU via our instrumented code, the methods based on ACE and Modified ACE were found not very useful because of their high round-off errors. Although not investigated, it is possible that these methods could be improved, for instance, by aggregating close-valued poles into a single pole value, avoiding some cancellation error problems. Computation of the jump probabilities in AU via uniformization resulted in small round-off errors which were, in fact, smaller than those for SU for many time points of interest, since the number of operations performed was smaller.

Comparisons of computational efficiency of the methods were limited to LU and SU due to the severe round-off errors exhibited by ACE and Modified ACE. With regard to these methods, it was found for short mission times in extended machine-repairman models that AU was orders of magnitude better than SU. For large time points, however, AU is less attractive than SU, due to its higher overall complexity. The two algorithms thus exhibit a "crossover point" where, on one side, AU is better than SU, and on the other, SU better than AU. The time crossover point t_T was found to be very close to the the operation crossover point t_F, demonstrating the viability of the operation counts. Studies of varying size models showed that AU's performance improves significantly with larger state spaces, as the reduction in the number of matrix vector multiplications becomes more important. Finally, it was shown that the computational cost of AU is more sensitive to the desired accuracy than is SU. Thus choosing a lower accuracy with AU will reduce run time more significantly than in SU. Furthermore, like SU, AU with uniformization to compute the jump probabilities is stable and exhibits very low round-off errors for the models studied.

These results suggest applications that could benefit directly from adaptive uniformization and, due to the crossover point, suggest that AU could profitably be combined with other methods. In particular, a solver may be buildable that estimates the crossover point, and selects the best method given the mission time. Thus multiple methods might be combined to achieve more efficient solutions than any single method (see [15]). Further study is necessary to determine if and when these hybrid approaches would be advantageous.

APPENDIX A EMR Model Documentation

In order to derive the continuous-time Markov chains necessary to test the algorithms, the modeling tool *UltraSAN* [2] was used. From a graphical description, *UltraSAN* generates the state space of a CTMC from the SAN description.

The SAN for the EMR model is given in Figure A.2, and consists of five kinds of components: *places* (circles), *timed actvities* (elongated ovals), an *instantaneous activity* (vertical bar), *input gates* (triangles pointing left), and *output gates* (triangles pointing right). The activities represent delays and events in the system. Timed activities require a non-negligible time to complete, and for a SAN to represent a CTMC, this time must be exponentially distributed. Instantaneous activities require no time, and complete upon activation. Both of these components can have *cases*, represented graphically by attached small bubbles. They allow different events to occur upon activity completion with some user-defined probability.

Places hold tokens. In the EMR model, the tokens in the place `working` represent operational units. The number of tokens in a place is called the *marking* of the place. The *MARK()* macro returns the marking of a place, and is an extension of the *C* code that is used to define that attributes of the SAN components.

Input gates have an enabling function, or predicate, and a function. The predicate is defined by the places to which it is connected, and when true, enables the activities to which it is connected. Upon activity completion, the function is executed, allowing changes to markings of connected places. Output gates have only a function, which is executed when its governing activity completes.

The following three appendices give documentation for the three example models based on the above.

Documentation for Composed Model: *EMR.*

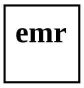

Figure A.1: Composed Model: *EMR*

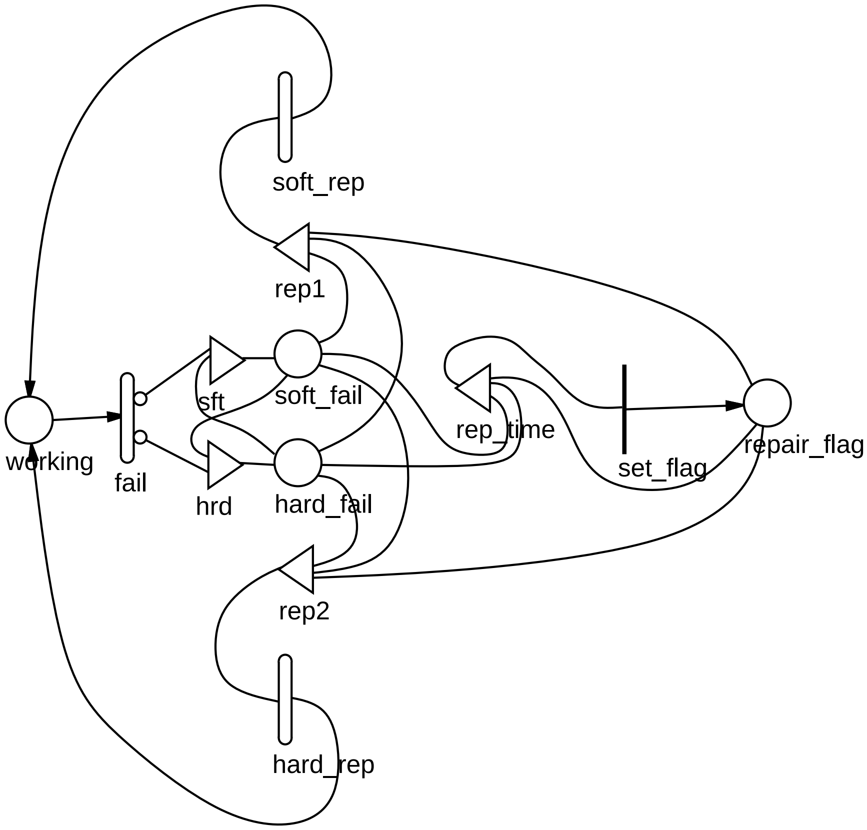

Figure A.2: SAN Model: *emr*

The places in Figure A.2 represent the following:

working Operational units.

soft_fail Units that have failed via a soft failure.

hard_fail Units that have failed via a hard failure.

repair_flag One if in a state of repair, zero otherwise.

Table A.1: Initial Non-zero Markings for SAN Model *emr*

Place	*Marking*
working	*20*

Table A.2: Activity Time Distributions for SAN Model *emr*

Activity	*Distribution*	*Parameter values*
fail	exponential	
	rate	*1.0* $*$ *MARK*(*working*)
hard_rep	exponential	
	rate	*800.0* $*$ *MARK*(*hard_fail*)
set_flag	instantaneous	
soft_rep	exponential	
	rate	*1000.0* $*$ *MARK*(*soft_fail*)

Table A.3: Activity Case Probabilities for SAN Model *emr*

Activity	*Case*	*Probability*
fail	1	*.5*
	2	*.5*

Table A.4: Input Gate Definitions for SAN Model *emr*

Gate	*Definition*
rep1	Predicate $MARK(soft\_fail)$ && $MARK(repair\_flag)$
	Function $MARK(soft\_fail) --;$ *if* $((MARK(soft\_fail) + MARK(hard\_fail)) == 0)$ $MARK(repair\_flag) = 0;$
rep2	Predicate $MARK(hard\_fail)$ && $MARK(repair\_flag)$
	Function $MARK(hard\_fail) --;$ *if* $((MARK(soft\_fail) + MARK(hard\_fail)) == 0)$ $MARK(repair\_flag) = 0;$
rep_time	Predicate $(MARK(soft\_fail) + MARK(hard\_fail) > 9)$ && $!MARK(repair\_flag)$
	Function *identity*

Table A.5: Output Gate Definitions for SAN Model *emr*

Gate	*Definition*
hrd	*if* $((MARK(soft\_fail) + MARK(hard\_fail)) == 19)$ { $MARK(soft\_fail) = 0;$ $MARK(hard\_fail) = 0;$ } *else* $MARK(hard\_fail) ++;$
sft	*if* $((MARK(soft\_fail) + MARK(hard\_fail)) == 19)$ { $MARK(soft\_fail) = 0;$ $MARK(hard\_fail) = 0;$ } *else* $MARK(soft\_fail) ++;$

APPENDIX B FTPCS Model Documentation

This appendix gives the model documentation for fault-tolerant parallel computer system with both perfect and non-perfect coverage.

B.1 FTPCS with Perfect Coverage

Documentation for Composed Model: *FTPCS.*

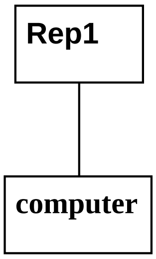

Figure B.1: Composed Model: *FTPCS*

The composed model for the FTPCS, Figure B.1, contains a *Rep* node. This allows replication of a submodel, and can reduce the resulting state space size by exploiting symmetries [27]. Interaction occurs through *common places*, as listed in Table B.1. Making places `repair_flag` and `comp_failed` common allows all computers in the set to know when repair is taking place and when the system has failed.

Table B.1: Composed Model Rep Node Definitions for Project *FTPCS*

Node	Reps	Common Places
Rep1	2	repair_flag
		comp_failed

The places in Figure B.2 represent the following:

comp_failed Failed computers.

cpus Operational CPUs.

errhandlers Operational error handler units.

inter Operational memory interface units.

ioports Operational I/O ports.

mems Operational memory boards.

mem_fail Failed memory board.

ram Operational RAM chips.

repair_flag One if system under repair, zero otherwise.

Table B.2: Initial Non-zero Markings for SAN Model *computer*

Place	*Marking*
cpus	*4*
errhandlers	*3*
inter	*2*
ioports	*3*
mems	*3*
ram	*40*

Table B.3: Activity Time Distributions for SAN Model *computer*

Activity	*Distribution*	*Parameter values*
cpu_fail	exponential	
	rate	$0.0052596 * MARK(cpus)$
err_fail	exponential	
	rate	$0.0017532 * MARK(errhandlers)$
inter_fail	exponential	
	rate	$0.0017532 * MARK(inter)$
io_fail	exponential	
	rate	$0.0052596 * MARK(ioports)$
mem_fail	instantaneous	
ram_fail	exponential	
	rate	$0.0008766 * MARK(ram)$
repair	exponential	
	rate	*1000.0*

Table B.4: Input Gate Definitions for SAN Model *computer*

Gate	*Definition*
IG_repair	<u>Predicate</u> $MARK(repair\_flag)$ && $!MARK(errhandlers)$ && $(MARK(comp\_fail) < 2)$
	<u>Function</u> *identity*

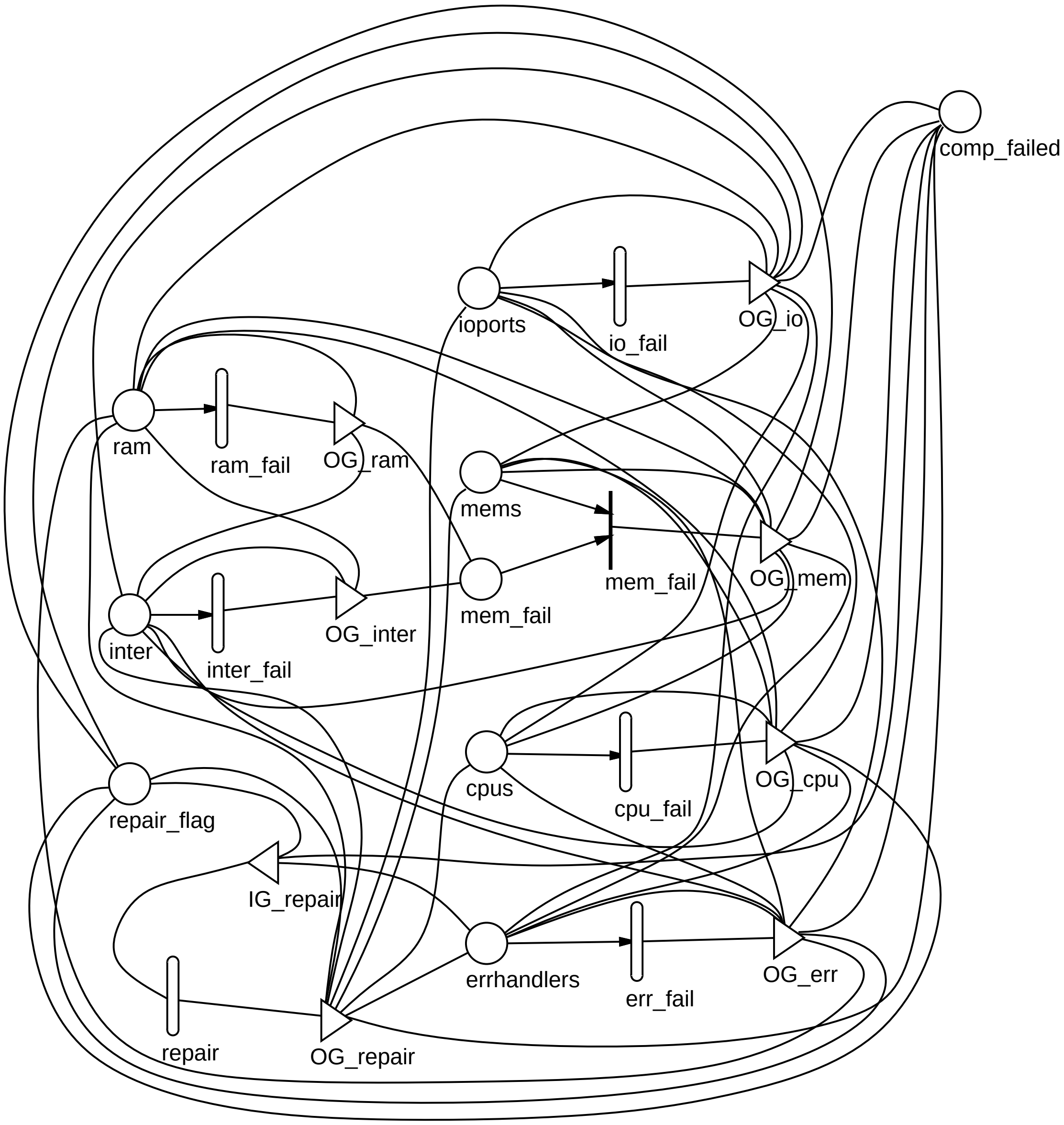

Figure B.2: SAN Model: *computer*

Table B.5: Output Gate Definitions for SAN Model *computer*

Gate	*Definition*
OG_cpu	*if (MARK(cpus) < 2)* { MARK(ram) = 0; MARK(inter) = 0; MARK(ioports) = 0; MARK(mems) = 0; MARK(cpus) = 0; MARK(errhandlers) = 0; MARK(comp_failed) ++; if (MARK(comp_failed) > 0) MARK(repair_flag) = 1; }
OG_err	*if (!MARK(errhandlers))* { MARK(ram) = 0; MARK(inter) = 0; MARK(ioports) = 0; MARK(mems) = 0; MARK(cpus) = 0; MARK(errhandlers) = 0; MARK(comp_failed) ++; if (MARK(comp_failed) > 0) MARK(repair_flag) = 1; }
OG_inter	*if (!MARK(inter))* { MARK(ram) = 40; MARK(inter) = 2; MARK(mem_fail) = 1; }
OG_io	*if (!MARK(ioports))* { MARK(ram) = 0; MARK(inter) = 0; MARK(ioports) = 0; MARK(mems) = 0; MARK(cpus) = 0; MARK(errhandlers) = 0; MARK(comp_failed) ++; if (MARK(comp_failed) > 0) MARK(repair_flag) = 1; }

Table B.6: Output Gate Definitions for SAN Model *computer*

Gate	*Definition*
OG_mem	*if* (*MARK*(*mems*) < *2*) { *MARK*(*ram*) = *0;* *MARK*(*inter*) = *0;* *MARK*(*ioports*) = *0;* *MARK*(*mems*) = *0;* *MARK*(*cpus*) = *0;* *MARK*(*errhandlers*) = *0;* *MARK*(*comp_failed*) + +*;* *if* (*MARK*(*comp_failed*) > *0*) *MARK*(*repair_flag*) = *1;* }
OG_ram	*if* (*MARK*(*ram*) < *39*) { *MARK*(*ram*) = *40;* *MARK*(*inter*) = *2;* *MARK*(*mem_fail*) = *1;* }
OG_repair	*MARK*(*ram*) = *40;* *MARK*(*inter*) = *2;* *MARK*(*ioports*) = *3;* *MARK*(*mems*) = *3;* *MARK*(*cpus*) = *4;* *MARK*(*errhandler*) = *3;* *MARK*(*comp_failed*) − −*;* *if* (*!MARK*(*comp_failed*)) *MARK*(*repair_flag*) = *0;*

B.2 FTPCS with Non-Perfect Coverage

Documentation for Composed Model: *FTPCS_npc.*

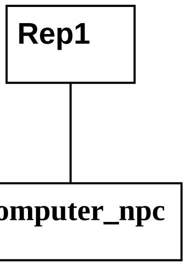

Figure B.3: Composed Model: *FTPCS_npc*

Table B.7: Composed Model Rep Node Definitions for Project *FTPCS_npc*

Node	**Reps**	**Common Places**
Rep1	1	comp_failed
		repair_flag

The FTPCS model with non-perfect coverage has the same set of places with the same meanings as the FTPCS model. The difference in this model is that most activities have cases to represent the coverage probabilities, as is evident in Figure B.4.

Table B.8: Initial Non-zero Markings for SAN Model *computer_npc*

Place	*Marking*
cpus	*4*
errhandlers	*3*
inter	*2*
ioports	*3*
mems	*3*
ram	*40*

Table B.9: Activity Time Distributions for SAN Model *computer_npc*

Activity	*Distribution*	*Parameter values*
cpu_fail	exponential	
	rate	$0.0052596 * MARK(cpus)$
err_fail	exponential	
	rate	$0.0017532 * MARK(errhandlers)$
inter_fail	exponential	
	rate	$0.0017532 * MARK(inter)$
io_fail	exponential	
	rate	$0.0052596 * MARK(ioports)$
mem_fail	instantaneous	
ram_fail	exponential	
	rate	$0.0008766 * MARK(ram)$
repair	exponential	
	rate	*1000.0*

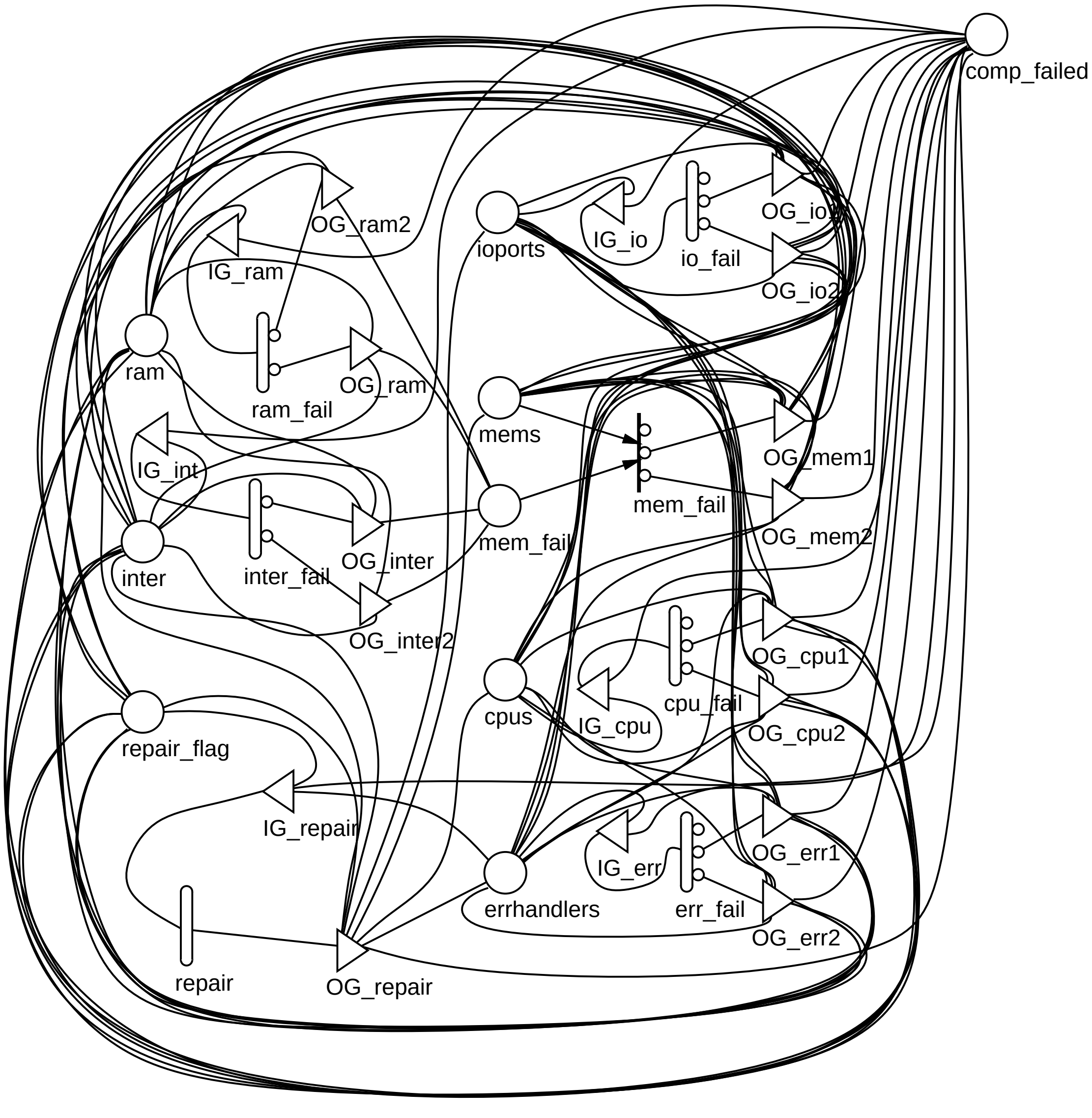

Figure B.4: SAN Model: *computer_npc*

Table B.10: Activity Case Probabilities for SAN Model *computer_npc*

Activity	*Case*	*Probability*
cpu_fail	1	*if (MARK(cpus) > 2)* *return(0.995);* *else* *return(ZERO);*
	2	*if (MARK(cpus) > 2)* *return(0.00475);* *else* *return(0.95);*
	3	*if (MARK(cpus) > 2)* *return(0.00025);* *else* *return(0.05);*
err_fail	1	*if (MARK(errhandlers) > 1)* *return(0.99);* *else* *return(ZERO);*
	2	*if (MARK(errhandlers) > 1)* *return(0.0095);* *else* *return(0.95);*
	3	*if (MARK(errhandlers) > 1)* *return(0.0005);* *else* *return(0.05);*
inter_fail	1	*0.998*
	2	*0.002*
io_fail	1	*if (MARK(ioports) > 1)* *return(0.99);* *else* *return(ZERO);*
	2	*if (MARK(ioports) > 1)* *return(0.0095);* *else* *return(0.95);*
	3	*if (MARK(ioports) > 1)* *return(0.0005);* *else* *return(0.05);*

Table B.11: Activity Case Probabilities for SAN Model *computer_npc*

Activity	*Case*	*Probability*
mem_fail	1	*if (MARK(mems) > 2)* *return(0.95);* *else* *return(ZERO);*
	2	*if (MARK(mems) > 2)* *return(0.0475);* *else* *return(0.95);*
	3	*if (MARK(mems) > 2)* *return(0.0025);* *else* *return(0.05);*
ram_fail	1	*0.002*
	2	*0.998*

Table B.12: Input Gate Definitions for SAN Model *computer_npc*

Gate	*Definition*
IG_cpu	Predicate $MARK(cpus)$ && $(MARK(comp\_failed) < 2)$
	Function $MARK(cpus)--;$
IG_err	Predicate $MARK(errhandlers)$ && $(MARK(comp\_failed) < 2)$
	Function $MARK(errhandlers)--;$
IG_int	Predicate $MARK(inter)$ && $(MARK(comp\_failed) < 2)$
	Function $MARK(inter)--;$
IG_io	Predicate $MARK(ioports)$ && $(MARK(comp\_failed) < 2)$
	Function $MARK(ioports)--;$
IG_ram	Predicate $MARK(ram)$ && $(MARK(comp\_failed) < 2)$
	Function $MARK(ram)--;$
IG_repair	Predicate $MARK(repair\_flag)$ && $!MARK(errhandlers)$ && $(MARK(comp\_failed) < 2)$
	Function *identity*

Table B.13: Output Gate Definitions for SAN Model *computer_npc*

Gate	*Definition*
OG_cpu1	*MARK(ram) = 0;* *MARK(inter) = 0;* *MARK(ioports) = 0;* *MARK(mems) = 0;* *MARK(cpus) = 0;* *MARK(errhandlers) = 0;* *MARK(comp_failed)++;* *MARK(repair_flag) = 1;*
OG_cpu2	*MARK(ram) = 0;* *MARK(inter) = 0;* *MARK(ioports) = 0;* *MARK(mems) = 0;* *MARK(cpus) = 0;* *MARK(errhandlers) = 0;* *MARK(comp_failed) = 2;* *MARK(repair_flag) = 1;*
OG_err1	*MARK(ram) = 0;* *MARK(inter) = 0;* *MARK(ioports) = 0;* *MARK(mems) = 0;* *MARK(cpus) = 0;* *MARK(errhandlers) = 0;* *MARK(comp_failed)++;* *MARK(repair_flag) = 1;*
OG_err2	*MARK(ram) = 0;* *MARK(inter) = 0;* *MARK(ioports) = 0;* *MARK(mems) = 0;* *MARK(cpus) = 0;* *MARK(errhandlers) = 0;* *MARK(comp_failed) = 2;* *MARK(repair_flag) = 1;*
OG_inter	*if (!MARK(inter))* { *MARK(ram) = 40;* *MARK(inter) = 2;* *MARK(mem_fail) = 1;* }

Table B.14: Output Gate Definitions for SAN Model *computer_npc*

Gate	*Definition*
OG_inter2	*MARK(mem_fail) = 1;* *MARK(ram) = 40;* *MARK(inter) = 2;*
OG_io1	*MARK(ram) = 0;* *MARK(inter) = 0;* *MARK(ioports) = 0;* *MARK(mems) = 0;* *MARK(cpus) = 0;* *MARK(errhandlers) = 0;* *MARK(comp_failed) ++;* *MARK(repair_flag) = 1;*
OG_io2	*MARK(ram) = 0;* *MARK(inter) = 0;* *MARK(ioports) = 0;* *MARK(mems) = 0;* *MARK(cpus) = 0;* *MARK(errhandlers) = 0;* *MARK(comp_failed) = 2;* *MARK(repair_flag) = 1;*
OG_mem1	*MARK(ram) = 0;* *MARK(inter) = 0;* *MARK(ioports) = 0;* *MARK(mems) = 0;* *MARK(cpus) = 0;* *MARK(errhandlers) = 0;* *MARK(comp_failed) ++;* *MARK(repair_flag) = 1;*
OG_mem2	*MARK(ram) = 0;* *MARK(inter) = 0;* *MARK(ioports) = 0;* *MARK(mems) = 0;* *MARK(cpus) = 0;* *MARK(errhandlers) = 0;* *MARK(comp_failed) = 2;* *MARK(repair_flag) = 1;*
OG_ram	*if (MARK(ram) < 39)* { *MARK(ram) = 40;* *MARK(inter) = 2;* *MARK(mem_fail) = 1;* }

Table B.15: Output Gate Definitions for SAN Model *computer_npc*

Gate	*Definition*
OG_ram2	*MARK(mem_fail) = 1;* *MARK(ram) = 40;* *MARK(inter) = 2;*
OG_repair	*MARK(ram) = 40;* *MARK(inter) = 2;* *MARK(ioports) = 3;* *MARK(mems) = 3;* *MARK(cpus) = 4;* *MARK(errhandler) = 3;* *MARK(comp_failed) − −;* *if (!MARK(comp_failed))* *MARK(repair_flag) = 0;*

APPENDIX C DDA Model Documentation

Documentation for Composed Model: *DDA.*

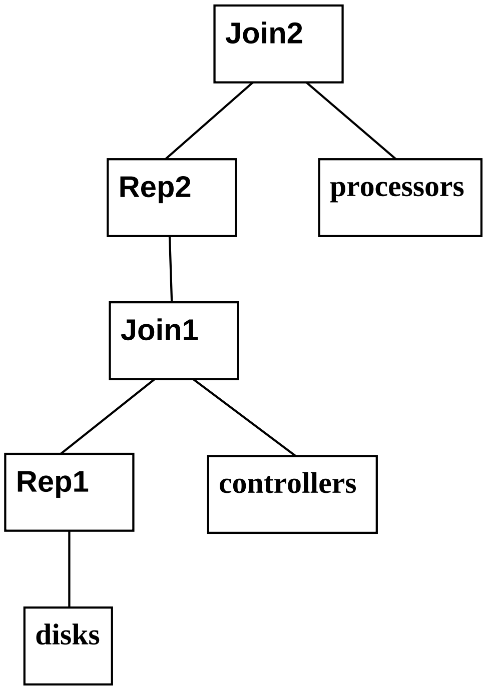

Figure C.1: Composed Model: *DDA*

In addition to *Rep* nodes, the DDA composed model in Figure C.1 has *Join* nodes. These allow two differing submodels to be joined with one or more common places. In this case `system_failure` is made common to all SANs.

Table C.1: Composed Model Rep Node Definitions for Project *DDA*

Node	**Reps**	**Common Places**
Rep1	2	system_failure
Rep2	2	system_failure

Table C.2: Composed Model Join Node Definitions for Project *DDA*

Node	**Common Places**		
Join1	Subtree	1	2
	system_failure	√	√
Join2	Subtree	1	2
	system_failure	√	√

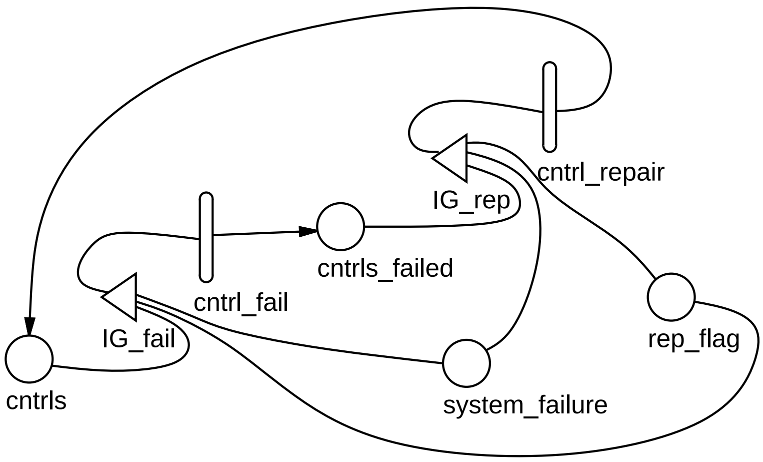

Figure C.2: SAN Model: *controllers*

The places in Figure C.2 represent the following:

cntrls Operational controllers.

cntrls_failed Failed controllers.

rep_flag One if in repair state, zero otherwise.

system_failure One if failed, zero otherwise.

Table C.3: Initial Non-zero Markings for SAN Model *controllers*

Place	*Marking*
cntrls	*3*

Table C.4: Activity Time Distributions for SAN Model *controllers*

Activity	*Distribution*	*Parameter values*
cntrl_fail	exponential	
	rate	*(1.0 / 2000.0) * MARK(cntrls)*
cntrl_repair	exponential	
	rate	*1.0*

Table C.5: Input Gate Definitions for SAN Model *controllers*

Gate	*Definition*
IG_fail	Predicate *!MARK(system_failure)*
	Function *MARK(cntrls) − −;* *if (MARK(cntrls) < 1)* *MARK(system_failure) = 1;* *if (MARK(cntrls) == 1)* *MARK(rep_flag) = 1;*
IG_rep	Predicate *!MARK(system_failure) && MARK(rep_flag)*
	Function *MARK(cntrls_failed) − −;* *if (!MARK(cntrls_failed))* *MARK(rep_flag) = 0;*

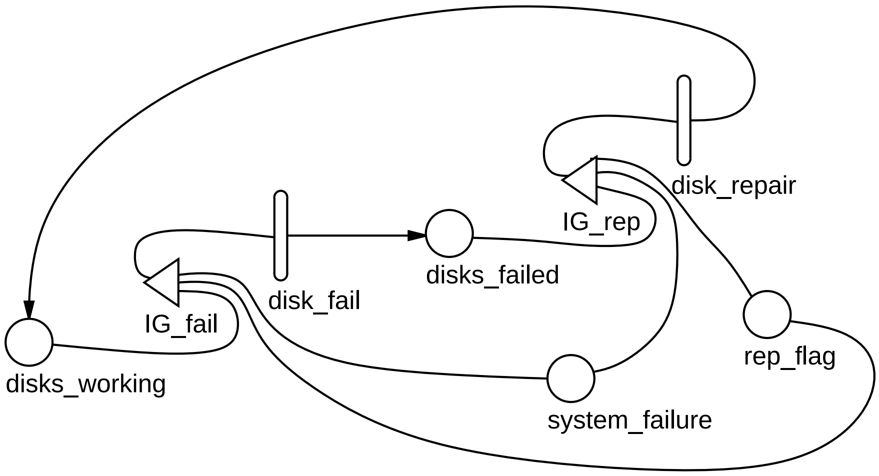

Figure C.3: SAN Model: *disks*

The places in Figure C.3 represent the following:

disks_working Operational disks.

disks_failed Failed disks.

rep_flag One if in repair state, zero otherwise.

system_failure One if failed, zero otherwise.

Table C.6: Initial Non-zero Markings for SAN Model *disks*

Place	*Marking*
disks_working	*5*

Table C.7: Activity Time Distributions for SAN Model *disks*

Activity	*Distribution*	*Parameter values*
disk_fail	exponential	
	rate	$(1.0\ /\ 6000.0) * MARK(disks\_working)$
disk_repair	exponential	
	rate	*1.0*

Table C.8: Input Gate Definitions for SAN Model *disks*

Gate	*Definition*
IG_fail	Predicate $!MARK(system\_failure)$
	Function $MARK(disks\_working) --;$ $if\ (MARK(disks\_working) < 3)$ $MARK(system\_failure) = 1;$ $if\ (MARK(disks\_working) == 3)$ $MARK(rep\_flag) = 1;$
IG_rep	Predicate $!MARK(system\_failure)\ \&\&\ MARK(rep\_flag)$
	Function $MARK(disks\_failed) --;$ $if\ (!MARK(disks\_failed))$ $MARK(rep\_flag) = 0;$

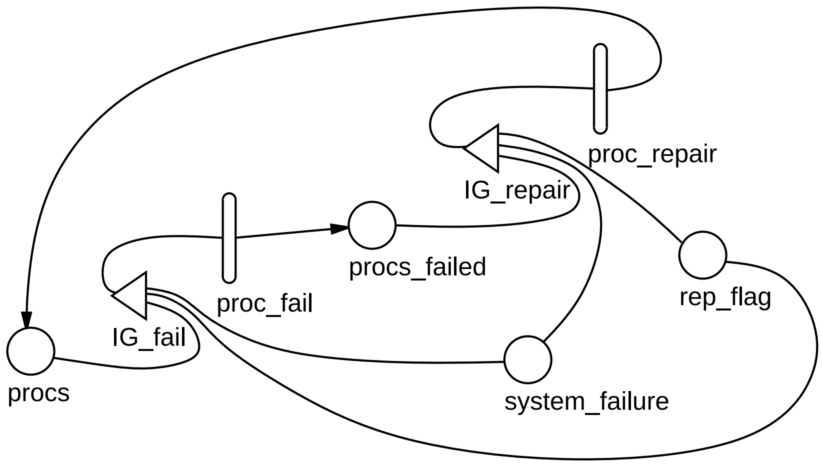

Figure C.4: SAN Model: *processors*

The places in Figure C.4 represent the following:

procs Operational processors.

procs_failed Failed processors.

rep_flag One if in repair state, zero otherwise.

system_failure One if failed, zero otherwise.

Table C.9: Initial Non-zero Markings for SAN Model *processors*

Place	*Marking*
procs	*3*

Table C.10: Activity Time Distributions for SAN Model *processors*

Activity	*Distribution*	*Parameter values*
proc_fail	exponential	
	rate	$(1.0\ /\ 2000.0) * MARK(procs)$
proc_repair	exponential	
	rate	*1.0*

Table C.11: Input Gate Definitions for SAN Model *processors*

Gate	*Definition*
IG_fail	Predicate *!MARK(system_failure)*
	Function *MARK(procs) − −;* *if (MARK(procs) < 1)* *MARK(system_failure) = 1;* *if (MARK(procs) == 1)* *MARK(rep_flag) = 1;*
IG_repair	Predicate *!MARK(system_failure) && MARK(rep_flag)*
	Function *MARK(procs_failed) − −;* *if (!MARK(procs_failed))* *MARK(rep_flag) = 0;*

APPENDIX D Solver Details

This appendix covers *UltraSAN* organization and where these solvers fit into the heirarchy, followed by a listing of the transient solver options, or program flags, in the first two sections. The next two sections detail solver inputs and outputs. Flow diagrams for the solvers are given in the final Appendix D section.

D.1 *UltraSAN* Organization

As can be seen in Figure D.1, the *UltraSAN* software package consists of four levels. These levels correspond to the process flow of modeling. The work done here fits into the solution portion of *UltraSAN*, at the bottom of Figure D.1. At this time, only standard uniformization is directly accessible from the *UltraSAN* control panel.

D.2 Solver Options

All the implementations are compatible and have the following options:

trs_su
trs_lu
trs_ace
trs_modace [-P project name] [-S study] [-E experiment] [-o output file] [-d debug file]
[-C count operations] [-a truncation ϵ] [-b LU jump process truncation ϵ_B]
[-v verbosity] [-t time point, -t time point, . . .]

[-P project name] The character string representing the project name. A name must be specified and no default is assumed.

[-S study] The character string representing the study name. A name must be specified and no default is assumed.

[-E experiment] The character string representing the experiment name. A name must be specified and no default is assumed.

[-o output file] A character string representing the output filename. The default is `stdout`.

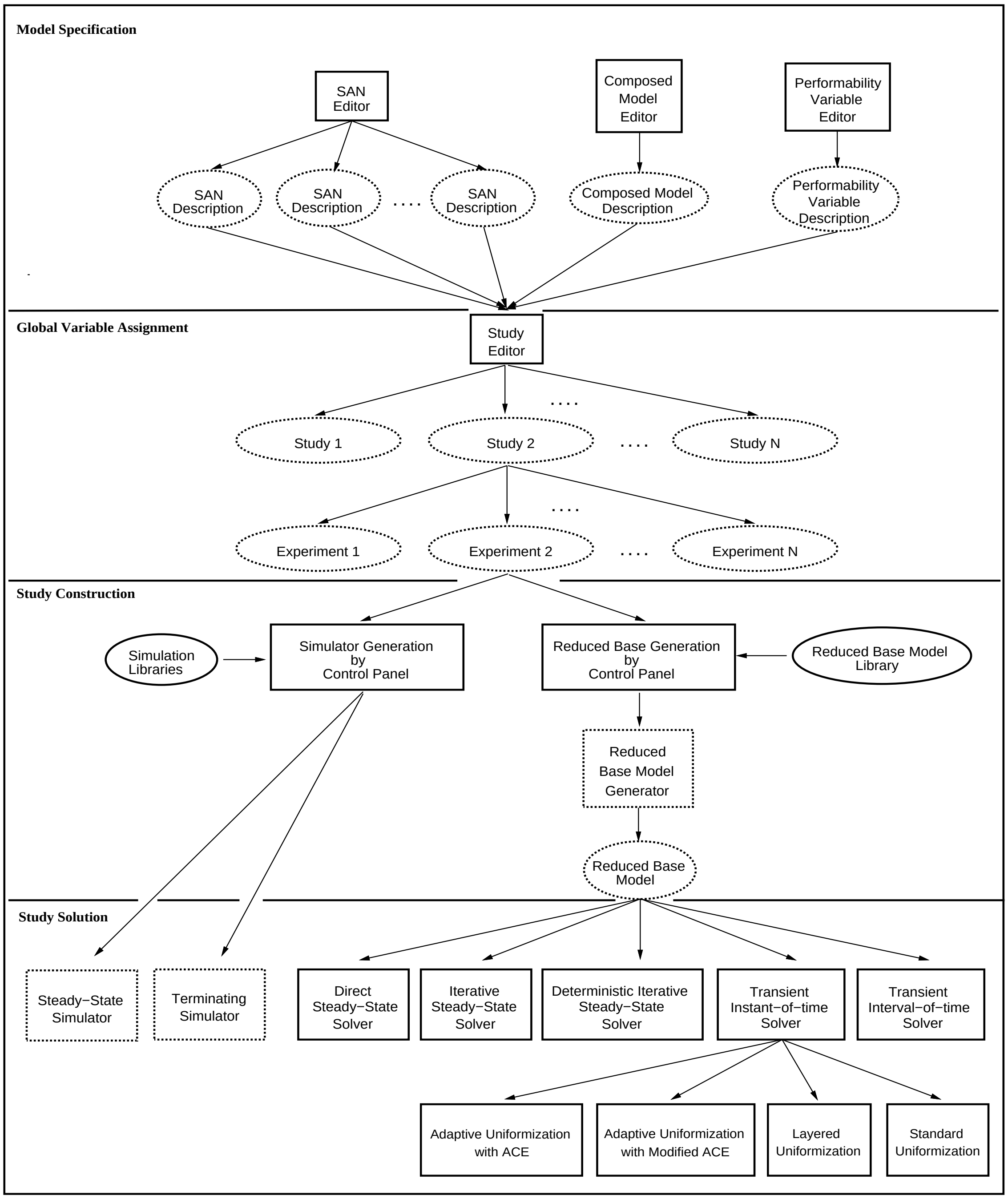

Figure D.1: *UltraSAN* Organization and Dataflow

[-d debug file] A character string representing the debug filename. The default is no debug file.

[-C count operations] This flag enables the operation counting and bound on round-off error. The output is modified by this flag to include operation counts and round-off error bounds on the peformance variables.

[-a truncation ϵ] An integer x such that $\epsilon = 10^{-x}$. x represents the number of digits of accuracy desired. The default is six. If x causes ϵ to be less than the machine accuracy, the machine accuracy becomes the default.

[-b LU jump process truncation ϵ_B] An integer y such that $\epsilon_B = 10^{-y}$. y determines the amount of truncation done during uniformization of the jump process. The default is $y = 14$.

[-v verbosity] An integer that determines how often status information is printed during computation. The integer corresponds to the number of interations between a status update, which consists of outputting the total number of iterations completed. When the -C option is on, AU versions output the jump process rates until convergence.

[-t time point] A double representing a time point of interest. Multiple time points can be specified, but as a minimum, at least one must be included as there is no default.

D.3 Solver Input

All four implementations use two input files. The first contains a representation of the CTMC. It is called a reduced base model [27] and is created by *UltraSAN*. The other contains the associated performance variable information. Both files are ASCII, and are pointed to by the project, study, and experiment name. Note that states are labeled $\{1, 2, \ldots\}$ rather than $\{0, 1, \ldots\}$ since 0 is used as a delimiter.

<u>Reduced Base Model File Format</u>

```
[unsigned long integer]        // initial state label
[unsigned long integer]        // total number of states
[unsigned long integer]        // state A label
[unsigned long integer]        // next state A1 from state A
[double]                       // rate to A1 from A
[unsigned long integer]        // next state A2 from state A
```

```
[double]                        // rate to A2 from A
         .
         .
         .
0                               // end of transitions from A
[unsigned long integer]         // state B label
[unsigned long integer]         // next state B1 from state B
[double]                        // rate to B1 from B
[unsigned long integer]         // next state B2 from state B
[double]                        // rate to B2 from B
        .
        .
        .
0                               // end of transitions from B
[unsigned long integer]         // state C label
        .
        .
        .
(continues until all states and their transitions have been defined)
```

Performance Variable Information File Format

```
[character string]              // performance variable name #1
[character string]              // performance variable name #2
        .
        .
        .
[character string]              // performance variable name #n
[unsigned long integer]         // state A label
[double]                        // impulse reward for state A, pv #1
[double]                        // rate reward for state A, pv #1
[double]                        // impulse reward for state A, pv #2
[double]                        // rate reward for state A, pv #2
        .
        .
        .
[double]                        // impulse reward for state A, pv #n
[double]                        // rate reward for state A, pv #n
[unsigned long integer]         // state B label
[double]                        // impulse reward for state B, pv #1
[double]                        // rate reward for state B, pv #1
[double]                        // impulse reward for state B, pv #2
[double]                        // rate reward for state B, pv #2
        .
        .
```

```
        .
[double]                    // impulse reward for state B, pv #n
[double]                    // rate reward for state B, pv #n
[unsigned long integer]     // state C label
        .
        .
        .
(continues until all states and their rewards have been defined)
```

D.4 Solver Output

The following sample output came from executing:

```
trs_lu -PEMR50 -Sstudy1 -Eexp1 -C -a12 -t0.6

***********************************************************************
                      UltraSAN TRANSIENT SOLVER

                      TRANSIENT SOLVER RESULTS
***********************************************************************

Project name: EMR50
Study name: study1
Experiment name: exp1

Global variable settings for this run:
No global variables in the model

Verbosity: 0
Accuracy: 1.000000e-12
BP Accuracy: 1.000000e-14
Time points: 0.600000


Computation Method: Layered Uniformization
Computation Time (seconds): 1134
User Time (seconds): 1129
System Time (seconds): 5
Rate of Poisson process: 49001.000000
Number of states in process: 2095
Number of states with reward: 2095
Number of time points: 1
```

```
     Time Point       # Iterations      Trunc. Error
   ---------------   ---------------   ---------------
        0.600000             11413      9.986456e-13

-------------------------------------------------------------------------

Performance variable :  Reliability

Time                 :  0.600000

Mean                     :  1.000000e+00
Bound on roundoff error  :  4.553923e-14

Variance                 :  1.013634e-12
Bound on roundoff error  :  1.368397e-13

Plot files: exp1.trs.Reliability.0.600.pdf.splot
            exp1.trs.Reliability.0.600.PDF.splot

-------------------------------------------------------------------------

Performance variable :  Repair

Time                 :  0.600000

Mean                     :  8.312345e-08
Bound on roundoff error  :  5.596571e-19

Variance                 :  8.312345e-08
Bound on roundoff error  :  5.596756e-19

Plot files: exp1.trs.Repair.0.600.pdf.splot
            exp1.trs.Repair.0.600.PDF.splot

-------------------------------------------------------------------------

Performance variable :  Avg. # Working

Time                 :  0.600000

Mean                     :  2.744060e+01
Bound on roundoff error  :  1.239478e-12
```

```
Variance                 :  1.238093e+01
Bound on roundoff error  :  1.025136e-10

Plot files: exp1.trs.Avg----Working.0.600.pdf.splot
            exp1.trs.Avg----Working.0.600.PDF.splot

--------------------------------------------------------------------------

Operation Area     Adds         Subs         Mults        Divs
-------------- ------------ ------------ ------------ ------------
     INIT      0.000000e+00 6.588000e+03 0.000000e+00 0.000000e+00
     JPR       1.115290e+08 1.311986e+07 1.246488e+08 7.201500e+04
     PQ        6.227000e+03 0.000000e+00 0.000000e+00 2.755100e+04
     MVM       9.869248e+07 0.000000e+00 9.869248e+07 2.000000e+00
     SOL       2.390814e+07 0.000000e+00 2.390814e+07 0.000000e+00
     PERF      1.257000e+04 3.000000e+00 1.885800e+04 0.000000e+00
               ------------ ------------ ------------ ------------
Totals:        2.341485e+08 1.312645e+07 2.472683e+08 1.121380e+05

--------------------------------------------------------------------------
```

D.5 Solver Flow Diagrams

Figure D.2 outlines the the code structure for SU, while that of AU is shown in Figure D.3.

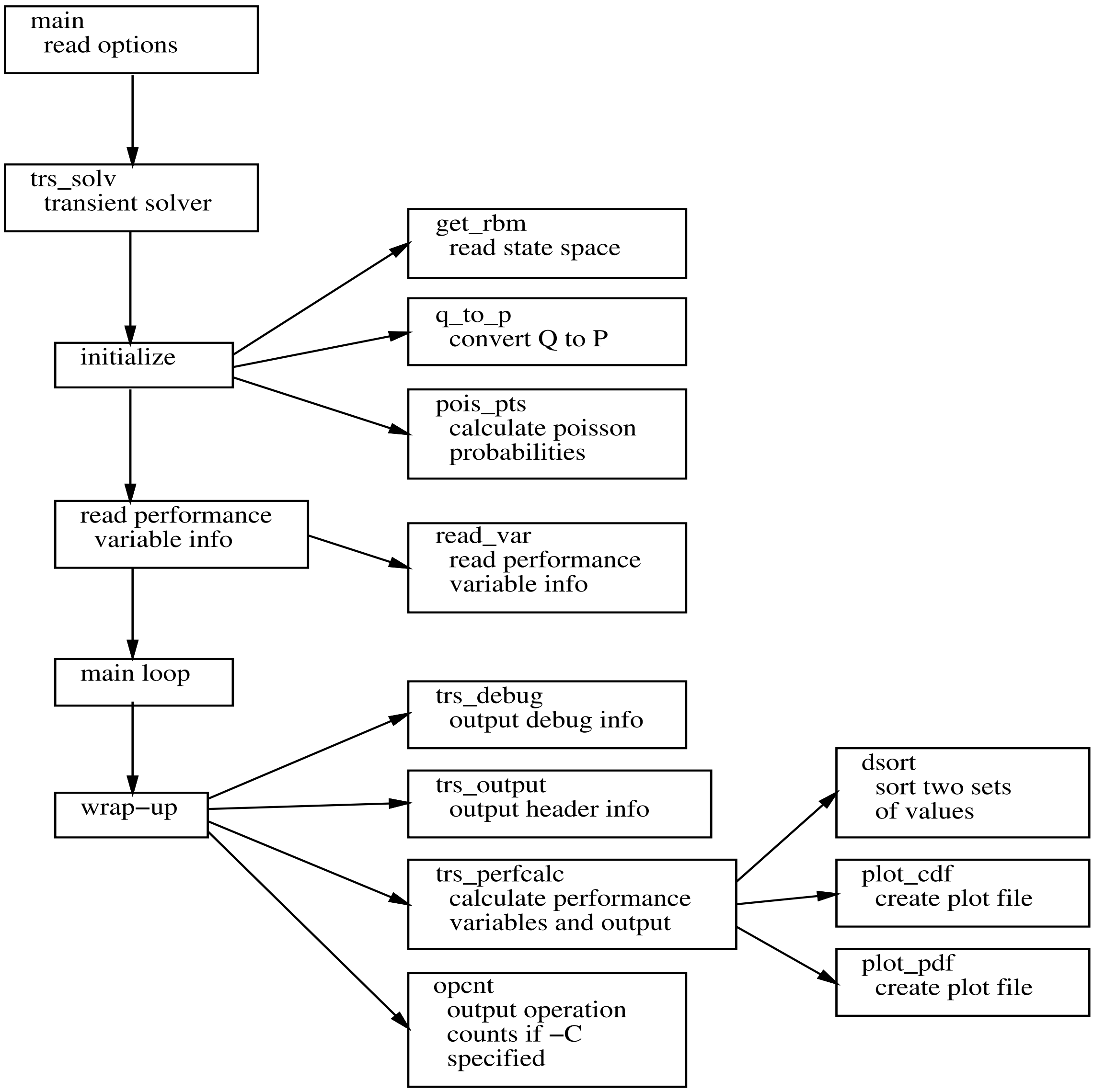

Figure D.2: Program Flow among Modules for SU

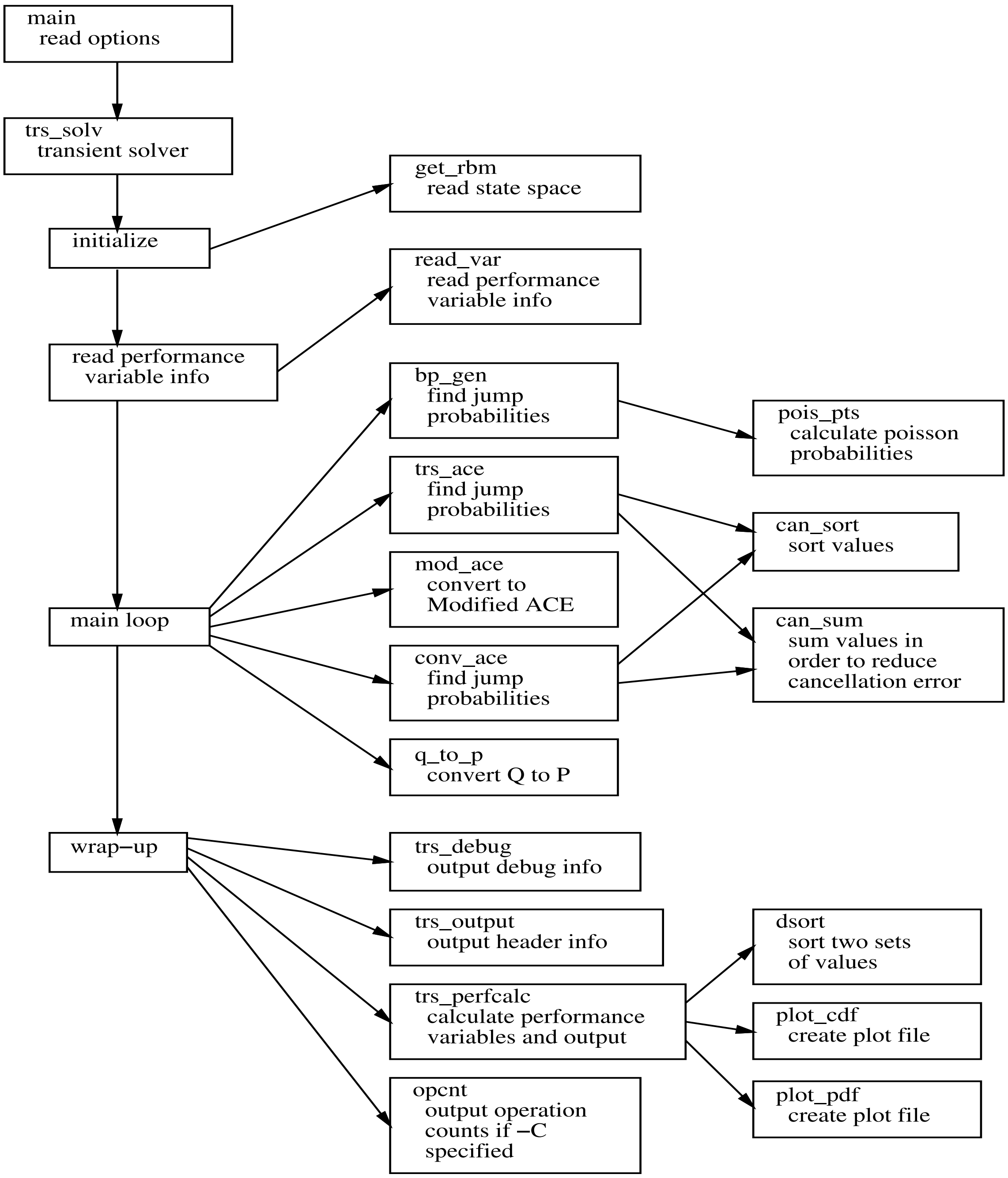

Figure D.3: Program Flow among Modules for AU

REFERENCES

[1] E. Çinlar, *Introduction to Stochastic Processes*, Prentice-Hall, Englewood Cliffs, 1975.

[2] J.A. Couvillion, R. Freire, R. Johnson, W.D. Obal II, M.A. Qureshi, R. Rai, W.H. Sanders and J. Tvedt, "Performability Modeling with UltraSAN," *IEEE Software*, **8**, (Sept. 1991), pp. 69-80.

[3] J. Dunkel and H. Stahl, "On the Transient Analysis of Stiff Markov Chains," *Proc. Third Working Conference on Dependable Computing for Critical Apllications*, Modello, Italy, Sept. 1992.

[4] W. Feller, *An Introduction to Probability Theory and Its Applications, Volume I and II*, New York, 1966.

[5] B.L. Fox and P.W. Glynn, "Computing Poisson Probabilities," *Comm. ACM* **31**, pp. 440-445, 1988.

[6] R. Geist and K.S. Trivedi, "Ultra-high Reliability Prediction for Fault-Tolerant Computer Systems," *IEEE Trans. on Comp.*, C-32(12), pp. 1118-1127, December 1983.

[7] A. Goyal, S. Lavenberg and K.S. Trivedi, "Probabilistic Modeling of Computer System Availability," *Annals of Operation Research*, **8**, pp. 285-306, 1987.

[8] W.K. Grassman, "Transient Solutions in Markovian queueing systems," *Computers & Operations Research* **4**, pp. 47-53, 1977.

[9] W.K. Grassman, "Finding Transient Solutions in Markovian Event Systems through Randomization," in *Numerical Solution of Markov Chains*, W.J. Stewart (Ed.), Marcel Dekker, New York, 1991.

[10] D. Gross and D.R. Miller, "The Randomization Technique as a Modelling Tool and Solution Procedure for Transient Markov Processes," *Operations Research*, **32:2**, pp. 343-361, March-April 1984.

[11] A. Jensen, "Markoff chains as an Aid in the Study of Markoff Processes," *Skand. Aktuarietidskrift.*, **36**, pp. 87-91, 1953.

[12] D. Lee, J. Abraham, D. Rennels and G. Gilley, "A Numerical Technique for the Evaluation of Large, Closed Fault-Tolerant Systems," In J.F. Meyer and R.D. Schlichting (Eds.), *Dependable Computing for Critical Applications*, Springer-Verlag, pp. 95-114, Wien, 1992.

[13] C. Lindemann, Private Communication, November 1993.

[14] C. Lindemann, M. Malhotra and K.S. Trivedi, "Numerical Methods for Reliability Evaluation of Closed Fault-tolerant Systems," *Technical Report*, Duke University, 1992.

[15] M. Malhotra, "A Unified Approach for Transient Analysis of Stiff and Non-Stiff Markov Models," Technical Report DUKE-CCSR-92-001, Center for Computer Systems Research, Duke University, 1992.

[16] C. Moler and C. Van Loan, "Nineteen Dubious Ways to Compute the Exponential of a Matrix," *SIAM Review*, **20:4**, pp. 801-836, October 1978.

[17] A.P.A. van Moorsel, "Performability Evaluation Concepts and Techniques," Ph.D. Thesis, University of Twente, 1993.

[18] A.P.A van Moorsel and W.H. Sanders, "Adaptive Uniformization," *Stochastic Models*, **10:3**, 1994.

[19] A.P.A. van Moorsel and W.H. Sanders, "Adaptive Uniformization: Technical Details," *PMRL Technical Report 93-4*, University of Arizona, 1992.

[20] A. Movaghar and J.F. Meyer, "Peformability Modeling with Stochastic Activity Networks," *Proceedings 1984 Real-Time Systems Symposium*, Austin, TX, December 1984.

[21] R.R. Muntz, E. de Souza e Silva and A. Goyal, "Bounding Availability of Repairable Computer Systems," *Performance Evaluation Rev.* **17**, pp. 29-38, 1989.

[22] V.F. Nicola, M.K. Nakayama, P. Heidelberger and A. Goyal, "Fast Simulation of Dependability Models with General Failure and Maintenance Processes," *Proc. of FTCS-20*, pp. 491-498, June 1990.

[23] R.A. Marie, A.L. Reibman and K.S. Trivedi, "Transient Analysis of Acyclic Markov Chains," *Perf. Eval.* **7**, pp. 175-194, 1987.

[24] A. Reibman, R. Smith and K.S. Trivedi, "Markov and Markov Reward Model Transient Analysis: An Overview of Numerical Approaches," *European Journal of Operational Research*, **40**, pp. 257-267, 1989.

[25] A. Reibman and K.S. Trivedi, "Numerical Transient Analysis of Markov Models," *Comput. Opns Res.* **15**, pp. 19-36, 1988.

[26] W.H. Sanders and L.M. Malhis, "Dependability Evaluation Using Composed SAN-Based Reward Models," *Journal of Parallel and Distributed Computing* **15**, pp. 238-254, 1992.

[27] W.H. Sanders and J.F. Meyer, "Reduced Base Model Construction Methods for Stochastic Activity Networks," *IEEE J. Selected Areas Comm.*, **9**, pp. 25-36, January 1991.

[28] N.C. Severo, "A Recursion Theorem on Solving Differential- Difference Equations and Applications to some Stochastic Processes," *J. Appl. Prob.* **6**, pp. 673-681, 1969.

[29] P.H. Sterbenz, *Floating-Point Computation*, Prentice-Hall, Englewood Cliffs, 1974.

[30] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Mass., 1987.

[31] K.S. Trivedi, *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*, Prentice-Hall, Englewood Cliffs, 1982.

[32] K.S. Trivedi, A. Reibman and R. Smith, "Transient Analysis of Markov and Markov Reward Models," *Computer Performance and Reliability*, North-Holland, pp. 535-544, 1988.