© 2011 Vinh V. Lam

A PATH-BASED FRAMEWORK FOR ANALYZING LARGE MARKOV MODELS

BY

VINH V. LAM

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Doctoral Committee:

Professor William H. Sanders, Chair, Director of Research
Professor Gul Agha
Professor José Meseguer
Professor David M. Nicol
Professor Peter Buchholz, Technische Universität Dortmund

ABSTRACT

Reliability and dependability modeling can be employed during many stages of analysis of a computing system to gain insights into its critical behaviors. To provide useful results, realistic models of systems are often necessarily large and complex. Numerical analysis of these models presents a formidable challenge because the sizes of their state-space descriptions grow exponentially in proportion to the sizes of the models. On the other hand, simulation of the models requires analysis of many trajectories in order to compute statistically correct solutions.

This dissertation presents a novel framework for performing both numerical analysis and simulation. The new numerical approach computes bounds on the solutions of transient measures in large continuous-time Markov chains (CTMCs). It extends existing path-based and uniformization-based methods by identifying sets of paths that are equivalent with respect to a reward measure and related to one another via a simple structural relationship. This relationship makes it possible for the approach to explore multiple paths at the same time, thus significantly increasing the number of paths that can be explored in a given amount of time. Furthermore, the use of a structured representation for the state space and the direct computation of the desired reward measure (without ever storing the solution vector) allow it to analyze very large models using a very small amount of storage.

Often, path-based techniques must compute many paths to obtain tight bounds. In addition to presenting the basic path-based approach, we also present algorithms for computing more paths and tighter bounds quickly. One resulting approach is based on the concept of path composition whereby precomputed subpaths are composed to compute the whole paths efficiently. Another approach is based on selecting important paths (among a set of many paths) for evaluation. Many path-based techniques suffer from having to evaluate many (unimportant) paths. Evaluating the important ones helps to compute tight bounds efficiently and quickly.

Furthermore, the basic path-based approach lays the foundation for constructing an efficient

simulation technique for solving Markov models. The technique is more efficient than traditional simulation techniques because it can eliminate redundant computations across trajectories and because it computes many trajectories at the same time. Moreover, the solutions it computes have better confidence levels than those computed by traditional techniques because it is based partially on exact numerical analysis of the models.

It is our thesis then that our approach can be used to solve large Markov models efficiently in terms of both storage and performance. In this dissertation, we present results to prove this claim.

To my family for their love and support.

ACKNOWLEDGMENTS

This dissertation was completed with the assistance and support of a number of people. I would like to thank my advisor, Professor William H. Sanders, for his technical advice, encouragement, support, and mentoring. This work would not have been possible without Professor Peter Buchholz. I would like to thank him for our productive collaboration and for hosting me in Dortmund, Germany. I would also like to thank Professors Gul Agha, José Meseguer, and David M. Nicol for being members of my dissertation committee. I appreciate their insightful questions, guidance, discussion, and supervision throughout the process of improving and completing this dissertation.

I would like to thank Ms. Jenny Applequist for her invaluable administrative and editorial assistance. Hardly anything would get done without her. Mr. Tod Courtney was instrumental in organizing, maintaining, and improving the Möbius code base. I thank him for helping me to understand how Möbius is organized and for acting as my sounding board throughout our many discussion sessions. I would like to thank also all of the past and present members of the Möbius projects: Amy Christensen, Graham Clark, David Daly, Dan Deavours, Salem Derisavi, Jay Doyle, Shravan Gaonkar, Mark Griffith, G. P. Kavanaugh, Michael McQuinn, Doug Obal, Eric Rozier, John Sowder, Aaron Stillman, Patrick Webster, and Alex Williamson. I am indebted as well to the other members of the PERFORM research group for providing a collegial, fun, and productive working environment: Vishu Gupta, Michael Ihde, Kaustubh R. Joshi, Sudha Krishnamurthy, Ryan Lefever, James Lyons, Jason Martin, HariGovind V. Ramasamy, Elizabeth Van Ruitenbeek, Sankalp Singh, and Fabrice Stevens.

Funding for my research was granted by a number of organizations. I would like to thank Pioneer Hi-Bred for their financial support and Motorola for funding the Motorola Center for High-Availability System Validation at the University of Illinois (under the umbrella of the Motorola Communications Center). I would like to thank also the National Science Foundation

(under Grant Nos. 9975019, 0086096, and CCR-0311616) for their financial support. Any opinions, findings, and conclusions or recommendations expressed in this material are mine and do not necessarily reflect the views of the sponsors.

Finally, I would like to thank my family for their encouragement, love, and support.

TABLE OF CONTENTS

LIST OF TABLES ix

LIST OF FIGURES x

CHAPTER 1 INTRODUCTION TO MARKOV CHAIN MODELING 1
1.1 Motivation for and Overview of Modeling 1
1.2 Overview of Solution Techniques 2
1.3 Mathematical Background on Numerical Solution Techniques 3
1.4 Related Work 5
1.4.1 Model Transformation Techniques 6
1.4.2 Model Representation Techniques 7
1.4.3 Numerical Techniques 8
1.5 Thesis Contribution 10
1.6 Dissertation Organization 11

CHAPTER 2 FOUNDATION FOR A PATH-BASED APPROACH 12
2.1 Motivation 12
2.2 Introduction 12
2.3 Transient Analysis of CTMCs Via Uniformization 14
2.4 Analyzable Class of Markovian Models and Their Underlying Matrix Structure . . . 15
2.5 Structured Path-based Approach 18
2.5.1 Conceptual Overview of the Approach 18
2.5.2 Numerical Analysis on Paths 19
2.5.3 Equivalence Relation Among Paths 22
2.6 Implementation Issues 22
2.7 Experimental Results 23
2.7.1 Model Description 24
2.7.2 Small-submodel-size Configuration 27
2.7.3 Medium-submodel-size Configuration 28
2.7.4 Large-submodel-size Configuration 29
2.8 Algorithmic Cost Analysis 30
2.9 Summary 31

CHAPTER 3 PATH-COMPOSITION APPROACH 36
3.1 Introduction 36
3.2 Review of the Structured Path-based Approach 37
3.2.1 Computation of Path Reward and Solution Bounds 38

3.2.2 Algorithm for Exploring Paths . . . 40
3.3 Algorithms for Path Composition and Path Selection . . . 41
3.3.1 The Path-Composition Algorithm . . . 42
3.3.2 Algorithms for Efficient Composition of Subpaths . . . 46
3.3.3 An Approach for Selecting Important Subpaths . . . 48
3.4 Numerical Results . . . 49
3.4.1 Model Description of the Double-Redundant Distributed Information Service System . . . 50
3.4.2 Model Description of the Media Multicast System . . . 52
3.4.3 Experimental Evaluation . . . 54
3.5 Summary . . . 56

CHAPTER 4 THE PATH-SELECTION APPROACH . . . 58
4.1 Introduction . . . 58
4.1.1 Motivation and Problem Statement . . . 58
4.1.2 Algorithmic Review of the Structured Path-based Approaches . . . 59
4.2 Technical Description of the Path-Selection Approaches . . . 60
4.2.1 Memory Management in the Path-Selection Approaches . . . 60
4.2.2 Path Selection Based on Reward Values of Paths . . . 61
4.2.3 Path Selection Based on Probability Values of Paths . . . 64
4.2.4 The Integrated Path-selection Approach . . . 65
4.2.5 Baseline Approach for Comparison with the Path-Selection Approaches . . . 69
4.3 Description of Experimental Models . . . 70
4.3.1 Availability and Reliability Models . . . 70
4.3.2 Performability Model . . . 71
4.4 Experimental Results and Discussion . . . 71
4.4.1 Availability Model . . . 72
4.4.2 Reliability Model . . . 83
4.4.3 Performability Model . . . 92
4.4.4 Discussion of Results . . . 97
4.5 Summary . . . 98

CHAPTER 5 PATH-BASED SIMULATION . . . 100
5.1 Introduction and Motivation . . . 100
5.2 Component-Level Analysis of Models . . . 101
5.3 The Path-Based Simulation Approach . . . 104
5.4 Evaluation of the Approach . . . 106
5.4.1 Model Description: Double-Redundant Distributed Information Service System . . . 106
5.4.2 Model Description: Media Multicast System . . . 107
5.4.3 Experimental Results . . . 107
5.5 Summary . . . 108

CHAPTER 6 CONCLUSION AND FUTURE WORK . . . 111

REFERENCES . . . 115

LIST OF TABLES

2.1 Mean corruption, failure, and repair rates for the submodels 26
2.2 Mean synchronized transition rates among submodels 26
2.3 Numerical results of reliability for the small-submodel-size configuration 27
2.4 Numerical results of availability for the small-submodel-size configuration 27
2.5 Numerical results of reliability for the medium-submodel-size configuration 28
2.6 Numerical results of availability for the medium-submodel-size configuration 29
2.7 Numerical results of reliability for the large-submodel-size configuration 29
2.8 Numerical results of availability for the large-submodel-size configuration 30

3.1 Numerical results for availability 54
3.2 Numerical results for reliability 54
3.3 Numerical results for the probability of buffer overflow 55
3.4 Numerical results for the probability of buffer flushing 55

4.1 Reliability model: Results reproduced from Table 7 in [1] 97

LIST OF FIGURES

1.1 General classification of solution techniques 4

2.1 SAN availability model of a distributed information service system 23
2.2 SAN reliability model of a distributed information service system 24
2.3 Convergence of the bounds for the reliability measure 32
2.4 Convergence of the bounds for the availability measure 33
2.5 Comparison of times to achieve tight bounds on the reliability measure. 34
2.6 Comparison of times to achieve tight bounds on the availability measure. 35

3.1 SAN availability model of a double-redundant distributed information service system 50
3.2 SAN reliability model of a double-redundant distributed information service system 51
3.3 SAN model of a media multicast system. 53
3.4 Performance evaluation of the path-composition approach 57

4.1 Algorithm for exploring paths in the structured path-based approach 59
4.2 Algorithm for computing transformed reward vectors. 62
4.3 Algorithm for path exploration based on path reward values. 62
4.4 Algorithm for selecting paths based on their reward values. 63
4.5 Algorithm for path exploration based on path probability values. 65
4.6 Path probability curves. 66
4.7 Path probability curves. 67
4.8 Integrated path-selection algorithm 68
4.9 Algorithm for baseline approach for evaluating paths. 70
4.10 Availability model: Probability-based approach at $t = 10.0$ 74
4.11 Availability model: Reward-based approach at $t = 10.0$ 75
4.12 Availability model: Integrated approach at $t = 10.0$ 76
4.13 Availability model: Combined graphs at $t = 10.0$ 77
4.14 Availability model: Probability-based approach at $t = 100.0$ 79
4.15 Availability model: Reward-based approach at $t = 100.0$ 80
4.16 Availability model: Integrated approach at $t = 100.0$ 81
4.17 Availability model: Combined graphs at $t = 100.0$ 82
4.18 Reliability model: Probability-based approach at $t = 10.0$ 84
4.19 Reliability model: Reward-based approach at $t = 10.0$ 85
4.20 Reliability model: Integrated approach at $t = 10.0$ 86
4.21 Reliability model: Combined graphs at $t = 10.0$ 87
4.22 Reliability model: Probability-based approach at $t = 100.0$ 88
4.23 Reliability model: Reward-based approach at $t = 100.0$ 89

4.24 Reliability model: Integrated approach at $t = 100.0$ 90
4.25 Reliability model: Combined graphs at $t = 100.0$ 91
4.26 Performability model: Probability-based approach at $t = 0.01$ 93
4.27 Performability model: Reward-based approach at $t = 0.01$ 94
4.28 Performability model: Integrated approach at $t = 0.01$ 95
4.29 Performability model: Combined graphs at $t = 0.01$ 96

5.1 Experimental results for the distributed information service system 109
5.2 Experimental results for the media multicast system 110

CHAPTER 1

INTRODUCTION TO MARKOV CHAIN MODELING

1.1 Motivation for and Overview of Modeling

Markov chain modeling is used in many fields spanning across such diverse areas as agricultural genetics, efficient environmental heating and cooling in buildings, and financial markets. In genetic engineering, for example, Markov models of genetic pathways are used to study their interconnections and their interactions in the process of producing proteins necessary for carrying out biological functions. In the financial sector, it has been used to analyze risks in trading strategies and futures markets. In computing systems, it has been used to locate bottlenecks and to assess additional benefits from increasing key resources. In reliability analysis, it has been used to estimate the mean time to failure of components in software, aerospace, and automotive systems just to name a few. Thus, Markov models and techniques for analyzing them are useful tools for gaining understanding of many aspects of everyday life, and they are used increasingly in many sophisticated applications.

The process of Markov chain modeling usually begins with a description of all possible discrete states that a physical system can occupy and a description of how the system transitions among the states in time. These states constitutes the *state space* of the model describing the system. If the transition from one state to another depends on only the current state, then the system is said to possess the *Markov property*. For Markov systems, this property is equivalent to specifying that the remaining, or sojourn, time that the system will spend in the current state is independent of the time it had already spent in that state. It can be shown easily (e.g. see [2]) that for discrete-time systems, this means that the sojourn time has a geometric distribution; for continuous-time systems, it has an exponential distribution. For this dissertation, we consider mainly continuous-time Markov chain models, and the exponential distribution is used implicitly. In a model description, the sojourn time is specified as a rate parameter (for an exponential

distribution) on a transition between two states. This specification is succinctly represented by a matrix, $\mathbf{Q}$, of rate values for all transitions among all states in the state space.

Given that a model starts in some initial state, the information that is most often sought from the model is the probability of occupying a certain state, or a subset of states, at some specified time in the future after the system has become operational. If the specified time is sufficiently long that the system has settled into a steady state, the obtained probability is referred to as the *stationary probability*. All other probabilities obtained before the system reaches steady state are referred to as *transient probabilities*[1].

In order to derive pertinent and readily usable information about the system being modeled, a *reward formalism* is used to specify numerical gains or losses when the system occupies some states or makes some transitions. For examples, by specifying a reward value in the unit of time for the repair state of a computing system, one can measure the expected time spent for repairing the system; and by specifying the number of jobs that can be processed per time unit in the working state, one can measure the expected throughput of the system. Thus, solving for the solution of a model is the process of analyzing the model and computing reward measures of interest from it.

1.2 Overview of Solution Techniques

After a model has been built and after reward values have been specified on it, the most important step in modeling is solving the model to obtain various interesting measures about the system being modeled. At a broad categorization level, there are two approaches for solving models: *numerical analysis* and *simulation*. The approach to use is chosen based on various characteristics about and the measures one is interested in computing from the models.

Numerical analysis solves a model by evaluating the state space of the stochastic process underlying the model. This approach is limited to solving only those models that can be represented by Markov chains. Furthermore, the state space of the model, equivalently the transition rate matrix $\mathbf{Q}$, is limited to a small finite size which can fit in the memory of a computer system on which the model will be analyzed. Although the research community is actively proposing effective techniques for compacting the state spaces and the transition rate matrices, the ultimate limitation of numerical analysis techniques is currently the representation

[1]There are models which do not have steady states.

of the solution vector, whose size is on the same order as that of the uncompacted state space. These limitations mean that solving sophisticated and realistic models, which are becoming more prevalent nowadays, is very difficult using traditional numerical techniques. On the other hand, for those models that are amenable to being solved by these techniques, their solutions can be obtained exactly instead of being statistically correct as obtained using simulation techniques. In addition, numerical analysis techniques can usually handle better events that occur rarely in a model.

Simulation solves a model by evaluating the trajectories of the stochastic process underlying the model. The state space is not needed for computing the solution, and because of this, this approach can solve a much bigger model. Moreover, it can solve more general models rather than just Markov ones. The solution that it computes, however, is not exact; it is only a statistical estimate of the exact solution. This means there is some confidence level that the estimated solution is within some confidence interval about the exact solution. Because the approach usually evaluates only a small subset of all trajectories to compute a solution, it may miss rare events that occur on those trajectories that it does not evaluate.

The solution techniques in both approaches are further classified as either transient or steady-state techniques. Especially for the numerical analysis approach, many steady-state techniques have been devised for computing steady-state solutions, but there are few satisfactory techniques for computing transient solutions of general models. A contribution of this dissertation is the presentation of both a numerical analysis technique and a simulation technique for computing transient solutions of Markov models. The numerical technique computes bounds around the exact solutions of the models, and it is capable of analyzing much larger models than traditional techniques can. For situations in which the numerical technique becomes less efficient (for example, even larger models or long-time horizon measures), the simulation technique can be employed to obtain estimated solutions. It is more efficient than traditional techniques because it can exploit redundant computations. In addition, it can compute solutions with better confidence intervals because it is based partially on exact numerical analysis.

1.3 Mathematical Background on Numerical Solution Techniques

Given a fully-specified Markov model, the underlying stochastic process can be constructed by exploring the reachability graph and storing the set of reachable states of the model and their

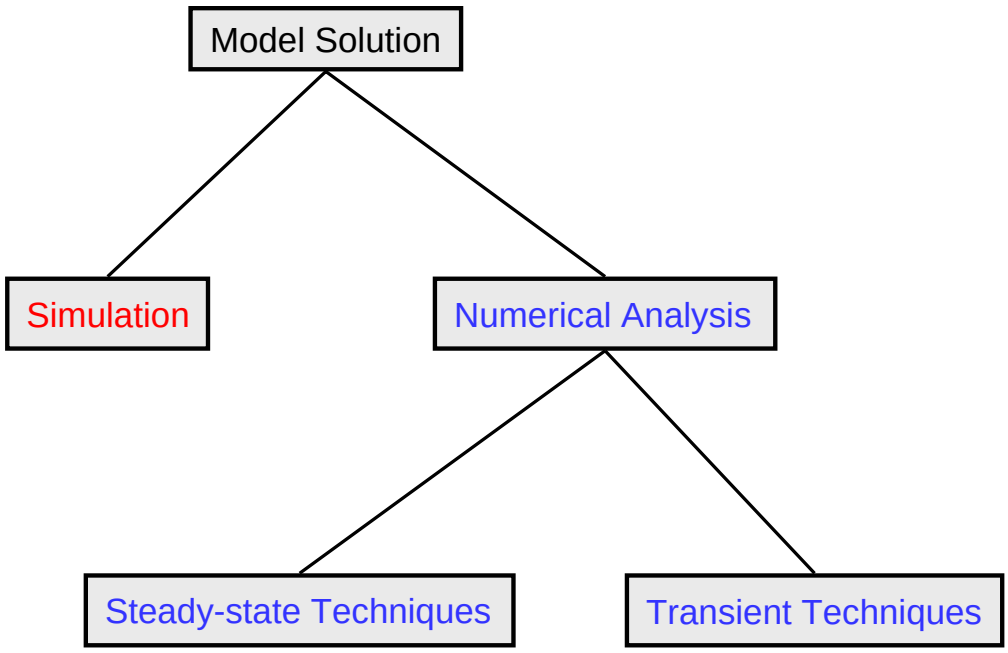


Figure 1.1: General classification of solution techniques. In this dissertation, we present a framework for unifying simulation and the transient numerical analysis technique.

corresponding transition rates. The resulting stochastic process is a continuous-time Markov chain (CTMC), $\{\mathcal{X}(t), t \geq 0\}$, having the transition rate matrix, or the *infinitesimal generator*, $\mathbf{Q} = [q_{ij}]$. Each element, q_{ij}, $i \neq j$, $1 \leq i,j \leq n$, specifies the rate of transition from state i to state j, and we define $q_{ii} = -\sum_{j \neq i} q_{ij}$. Note that this process has n states, and the dimension of $\mathbf{Q}$ is $n \times n$. The probability that the process is in some state i at time t is specified by $\boldsymbol{\pi}_i(t)$. The row vector $\boldsymbol{\pi}(t) = [\boldsymbol{\pi}_1(t)\ \boldsymbol{\pi}_2(t)\ \ldots\ \boldsymbol{\pi}_n(t)]$ specifies the probabilities that the process is in each of the states of the CTMC at time t, and it is called the *state probability vector*. Using the system of Kolmogorov differential equations[3], we can describe the behavior of the state probability vector as a function of time:

$$\frac{d}{dt}\boldsymbol{\pi}(t) = \boldsymbol{\pi}(t)\,\mathbf{Q} \tag{1.1}$$

with $\boldsymbol{\pi}(0)$ being the initial condition or the initial state vector.

Solving (1.1) for the steady-state solution involves setting the condition $\frac{d}{dt}\boldsymbol{\pi}(t) = \mathbf{0}$ and solving the linear algebraic system

$$\boldsymbol{\pi}\,\mathbf{Q} = \mathbf{0}, \quad \boldsymbol{\pi}\,\mathbf{e}^T = 1, \tag{1.2}$$

where $\mathbf{e}^T$ is the transposed row vector of ones. Many techniques for solving this problem have been proposed and examined in detail in the literature [4, 5]. There are, however, few effective techniques for computing transient solutions.

Solving for a transient solution requires solving a system of linear differential equations instead of a system of linear algebraic equations as required for computing a steady-state solution. The general solution for the system (1.1) is given by

$$\boldsymbol{\pi}(t) = \boldsymbol{\pi}(0)\, e^{\mathbf{Q}t}, \tag{1.3}$$

where $e^{\mathbf{Q}t}$ is the matrix exponential [6] defined by the Taylor series

$$e^{\mathbf{Q}t} = \sum_{k=0}^{\infty} \frac{(\mathbf{Q}t)^k}{k!}. \tag{1.4}$$

Although (1.3) yields an exact transient solution, computing it is quite difficult especially when the infinitesimal generator is a large matrix because it inundates the available memory on a computer. In the next section, we discuss related work that attempts to mitigate this problem.

1.4 Related Work

Devising effective techniques for solving ever larger models has always been a goal of researchers since the inception of the field of Markov model analysis. In this section, we briefly review a few of the related techniques that are effective for solving very large models[2]. Although there are techniques that overlap several classification categories, we classify these techniques mainly on how they manipulate the state spaces of the models.

[2]Models having product form solutions are not considered here.

1.4.1 Model Transformation Techniques

In the model transformation category, we include those techniques that modify the underlying state space of a model in order to mitigate the state-space explosion problem. The category encompasses both exact and approximate transformation of the state space. The techniques in the exact transformation category modify the state space in a way that the original one and the modified one are functionally equivalent with respect to some measure of interest. *Lumping* [7, 8, 9, 10] is an approach in this category that aggregates states that have the same transition rates to a set of lumpable states. Various lumping techniques exist to compute an optimal lumped state space when given the complete infinitesimal generator.

The decompositional approach, in general, approximates the original state space by utilizing the divide-and-conquer principle. In this approach, a large model is divided into small submodels that are solved independently. Their solutions are then composed to obtain the solution to the original model. The assumption made in this approach is that the submodels do not interact with each other so that they can be solved independently. Usually, this assumption does not hold in real models, and this gives rise to error. In the class of *nearly completely decomposable* (NCD) models, however, the submodels interact weakly, and this approach can be used to approximate the original model. Work on NCD models was originally carried out by Simon and Ando [11] and Courtois [12]. Later work extend their work to compute error bounds. A method that uses this approach to solve stiff models is time-scale decomposition [13] in which a decomposition of the state space is made based on fast and slow rates (which usually are considered to differ by several orders of magnitude). Other well-known methods that iteratively aggregate and disaggregate (IAD) states are found in [14, 15, 16, 17]. Many of these methods are used for computing stationary solutions because they iteratively solve the submodels first before composing their solutions. An exception is the splitting method in [18] that decomposes a cyclic Markov chain into subchains that are solved independently for a transient solution. This method, however, still has to deal with the large solution vector whose size is on the order of the infinitesimal generator. In [19], a method is proposed for addressing the problem of the large solution vector. Similar to other IAD methods, this one also computes stationary solutions.

1.4.2 Model Representation Techniques

The main idea in many of the techniques in the model representation category is the use of novel data structures and algorithms to represent efficiently the state space and the infinitesimal generator of a model. Two popular approaches in this category are based on decision diagrams and Kronecker representation. The goal of these approaches is to exploit redundancy in the representation of state information to achieve much more efficiency than sparse or list storage schemes can.

The approach of using decision diagrams is an outgrowth of the work on *binary decision diagrams* (BDD) [20]. BDD is an information encoding scheme that is based on a series of binary decisions. BDD and its derivative, multi-valued decision diagrams (MDD) [21], have been used successfully in the field of formal methods [22, 23] to generate and store very large state spaces. Later, an extension was made for storing matrices in a technique called *matrix diagrams* (MxD) [24]. Thus, MDD and MxD can be used to store the state space and the infinitesimal generator of a model, and they have been used very effectively for analyzing large models. For example, in [25], they were used together with a lumping technique to analyze models having state spaces that were several orders of magnitude larger than other techniques could analyze. This approach, however, cannot be used effectively yet for compact representation of the solution vector.

The Kronecker representation approach uses tensor algebraic operators to construct its representation of the infinitesimal generator. The approach employs the concept of automata network [5] to describe how automata in the network interact with each other. Each automaton is represented by a set of states and follows probabilistic rules that prescribe how it transitions from one state to another. In this setting, an automaton is the equivalence of a submodel, and the tensor operators describe how the automaton evolves and how it interacts with other automata.

Initial work on Kronecker representation was carried out by Plateau in [26]. The main idea of the approach is to store the infinitesimal generator, $\mathbf{Q}$, *implicitly*. That is, it is never generated or stored in full. Its values can be computed when needed by composing the submatrices corresponding to the submodels by means of the tensor operators. It should be noted that the total size of all the submatrices is much smaller than the size of the full $\mathbf{Q}$. In her pioneering work, Plateau used the slow-converging iterative Power method with the dense storage scheme for the submatrices, and the computation was done on the *potential* state space — a potentially

much larger superset of the reachable states in the actual state space. Later work improved upon her work by considering other faster iterative methods [27, 28, 29, 5], by considering sparse storage schemes [30], and by considering the overhead of the unreachable states in the potential state space [31, 32]. Similar to the approach of using the decision diagrams, this approach cannot be used to represent the solution vectors effectively.

1.4.3 Numerical Techniques

In this category, we include methods that relies mostly on numerical techniques in order to evaluate large models. Since (1.1) is a system of differential equations, it seems that methods from ordinary differential equations (ODE) are natural candidates to use to solve it. The field of ODE has a large amount of literature on many methods for solving differential equations. Here we summarize a few of the methods from [5] that have been considered for solving Markov models. Equation (1.1) has a family of solutions, and the particular one that solves a given model is computed from the initial condition of the model. The solution curve is computed by successive approximations to the curve. Techniques that are based on successive approximations must contend with the problem of accuracy. In addition, methods such as Euler and its variants must still represent and manipulate a system of equations whose size is on the order of the size of the state space. In fact, any method based on the evaluation of the Taylor series expansion of (1.3), of which Euler is one, is not efficient because it involves taking higher derivatives of the matrix exponential. Although the Runge-Kutta method and other multi-step methods such as the method of Adams and Milne avoid taking higher derivatives, they again have to represent and manipulate a system of equations whose size is as large as that of the state-space size.

Another method that has been considered for solving large sparse systems [5] is the Krylov subspace method. This method involves first building the Krylov subspace

$$K_m(\mathbf{Q}^T, v) \equiv \text{span}\{v, (\mathbf{Q}^T)v, \ldots, (\mathbf{Q}^T)^{m-1}v\}.$$

Then an approximation of the form $p_{m-1}(\mathbf{Q}^T)\ \boldsymbol{\pi}^T(t)$ to the matrix exponential $e^{\mathbf{Q}}\boldsymbol{\pi}^T(t)$ is sought, where $p_{m-1}(\mathbf{Q}^T)$ is a vector in the span of the Krylov subspace. Note, however, that the Krylov subspace contains vectors whose sizes are on the order of the size of the state space of a model. Thus, this method will have difficulty solving large models.

In order to avoid the memory complexity that many of the previous methods encounter in analyzing large models, several techniques [33, 34] propose the idea of generating a small set of states with high probabilities instead of the complete state space. The problem with this approach, however, is that the intermediate states to those in the set must still be generated. A similar idea that attempts to minimize the memory complexity is that of path analysis of the underlying Markov chain. In path-based analysis, the memory complexity is on the order of the length of a path. Thus, this approach has the potential for analyzing very large models, and we examine several of the related techniques here.

An important issue in any path-based technique is the computation of the probability of a path. In [35], a technique based on renewal argument is proposed, and it is restricted to analyzing paths having total outgoings rates that are distinct. Nicol and Palumbo proposed a technique in [36] that relaxed this restriction. In computing the probabilities of paths having slow transitions, they proposed using the uniformization technique [5]. This computation, however, requires the full infinitesimal generator. Moreover, the memory complexity of their technique depends on the lengths and fanouts of the paths. Because of this, their technique is limited to analyzing only a few failures in a model.

Some other notable early work on path-based analysis was carried out by de Souza e Silva and Gail in [37]. The storage and computation complexities of their technique, however, is combinatorial with the number of distinct rewards in a model. Donatiello and Grassi [38] improved upon this and presented an algorithm with polynomial complexity in the number of distinct rewards by combining uniformization and Laplace transform methods. Later, de Souza and Gail presented a significant improvement in their previous algorithm in [39]. In spite of all of these significant advances, however, these algorithms are limited to solving small models because their storage complexity is still polynomial with the number of states and transitions and they have to store the full transition matrix. In [40], Qureshi presented an algorithm having storage complexity that is linear in the number of states in a path. The algorithm uses the renewal argument as the basis of it formulation, and it was susceptible to numerical inaccuracy. The author suggested using extended precision arithmetic to alleviate this problem. A multi-precision library was used, and it was shown that this was very expensive. As a result, only a few, short paths can be analyzed using the algorithm.

1.5 Thesis Contribution

Modeling has been widely used, and it is increasingly used in quite sophisticated applications. Solving very large models have been the driving goal of the field of modeling and analysis for a long time. One of the major obstacles to overcome to achieve that goal has been to manage the storage complexity effectively. This dissertation makes the following contributions toward achieving that goal:

1. Development of a path-based framework from which both analytical and simulation techniques can be formulated for computing solutions to very large models. To the best of our knowledge, the framework is the first to unify both analytical and simulation techniques.

2. Development and implementation of an analytical approach for computing bounds on solutions of models that are several orders of magnitude larger than many current techniques can compute. Even when solving those large models (billion of states), the approach uses only tens of megabytes instead of gigabytes that many other techniques requires.

3. Formulation of a partial-order reduction relation for identifying and computing equivalent paths. Our work is the first to make use of the partial-order reduction technique for efficient analysis of Markov chains.

4. Formulation of a composition algorithm for computing large sets of paths efficiently. We believe our work is the first to propose the idea of path composition for efficient analysis Markov chains. As part of the contributions of the thesis, we also implemented an approach that exploits the algorithm's ability to eliminate redundant computations to evaluate many paths efficiently.

5. Formulation of an algorithm for identifying and computing important paths. The ability to identify important paths among a large set of paths and selectively evaluate only those important ones help in computing tight bounds quickly and efficiently. We also developed three path-selection approaches based on various criteria to choose the important paths and compared their performance in computing tight bounds.

6. Development and implementation of a path-based simulation approach that is more efficient than traditional simulation techniques. Simulating at a high level of abstraction, our approach is able to evaluate multiple trajectories at the same time.

1.6 Dissertation Organization

The rest of this dissertation is presented in the following order. In Chapter 2, we present the foundation for constructing the path-based approach. This chapter covers the basic mathematics needed for computing transient rewards. Then it proceeds to show the derivation that forms the basis for constructing and computing paths. The chapter concludes with the evaluation of the approach in solving an example model that is several orders of magnitude larger than what many other current techniques can evaluate.

In Chapter 3, we present the path composition algorithm. First, we discuss the concept of how paths are composed and provide the intuition for it. Then we provide the mathematical justification for it. In order to make the algorithm efficient, we present another algorithm for computing composable sets of paths. Next, show the derivation of complete equation for computing composed paths. Lastly, we evaluate the approach in solving several large models.

We present next the algorithm for selecting important paths in Chapter 4. Paths may be prioritized according to different criteria. We discuss how they are selected through three approaches. The first two approaches consider individually two factors that determine the priority of the paths. The third chooses dynamically between the factors at runtime. We then evaluate the performance of the approaches and compare them against that of the basic approach described in 2.

Then, in Chapter 5, we present the development of a path-based simulation approach. Since the approach simulates a model at the component level, it can sample multiple trajectories at the same time. We show how paths are sampled in approach and how solutions are computed.

Finally, in Chapter 6, we conclude with a few remarks. In this dissertation, we have built only the foundation and framework for analyzing models through the evaluation of paths. We discuss several ideas for future research.

This dissertation also includes two appendices. In Append A, we document all of the models we used to evaluate the approaches. In Appendix B, we explain the procedure for constructing models to be solved using our approaches.

CHAPTER 2

FOUNDATION FOR A PATH-BASED APPROACH

2.1 Motivation

Structured (a.k.a. symbolic) representation techniques of Markov models have, to a large extent, been used effectively for representing very large transition matrices and their associated state spaces. However, their success means that the largest space requirement encountered when analyzing these models is often the representation of their iteration and solution vectors. In this chapter, we present a new approach for computing bounds on solutions of transient measures in large continuous-time Markov chains (CTMCs). The approach extends existing path- and uniformization-based methods by identifying sets of paths that are equivalent with respect to a reward measure and related to one another via a simple structural relationship. This relationship allows us to explore multiple paths at the same time, thus significantly increasing the number of paths that can be explored in a given amount of time. Furthermore, the use of a structured representation for the state space and the direct computation of the desired reward measure (without ever storing the solution vector) allow us to analyze very large models using a very small amount of storage. In addition to presenting the method itself, we illustrate its use to compute the reliability and the availability of a large distributed information service system in which faults may propagate across subsystems.

2.2 Introduction

In performance and dependability analysis, transient measures are often computed by discrete event simulation or by numerical analysis of the CTMCs underlying the models. Both approaches have their drawbacks: simulation can only estimate results up to a certain accuracy because it relies on the observation of a stochastic process, while numerical analysis suffers from the problems associated with state-space explosion. Moreover, when the intervals to be simulated are

long and there is large variance in the transient measures being computed, many trajectories need to be analyzed, but no bounds can be placed on the accuracy of the obtained results. Even though statistical confidence intervals can be calculated in simulation, without bounds one cannot absolutely determine how close the obtained results are to the correct results. Numerical analysis, on the other hand, offers the potential to compute bounds when the exponential growth in state-space size, the associated transition matrix, and the solution vector can be managed efficiently.

Recently, a number of researchers have devised data structures and algorithms to manage the state space efficiently. One of the approaches represents the generator matrix of a CTMC in compact form as a *Kronecker* product of smaller component matrices [26, 5]. An interesting approach that successfully exploits data structures is based on *decision diagrams* and *matrix diagrams*, e.g., [24]. These techniques make it possible to analyze CTMCs that have state spaces that are several orders of magnitude larger than could previously be handled. In effect, they shift the bottleneck encountered during solution from the storage of the state space and transition matrix to the storage of the iteration and solution vectors (whose size is on the order of that of the state space).

In this chapter, we present a new path-based approach for computing the solutions of transient measures in large CTMCs. The approach extends existing path- and uniformization-based methods (e.g., [41, 42, 36]) by identifying sets of paths that are equivalent with respect to a reward measure and related to one another via a simple structural relationship. This relationship allows us to explore multiple paths at the same time, and thus significantly increases the number of paths that can be explored. Furthermore, the use of a structured representation for the state space and the direct computation of the desired reward measure (without ever storing the solution vector) allow us to analyze very large models using a modest amount of storage. In addition, due to the combined aspects of path and uniformization analysis, our approach can compute both lower and upper bounds on the solution.

This chapter proceeds as follows. In Section 2.3, we briefly review the transient analysis of reward measures for CTMCs via the uniformization method. Next, in Section 2.4, we describe the class of Markovian models that we intend to analyze and their underlying matrix structures. Then, in Section 2.5, we introduce our path-based approach utilizing the methods described in the previous two sections. In Section 2.6, we discuss issues related to the implementation of our

approach. In Section 2.7, we present some preliminary results, and in Section 2.8, we discuss the algorithmic costs of the approach. Finally, in Section 2.9, we conclude our presentation with a discussion of future work.

2.3 Transient Analysis of CTMCs Via Uniformization

Uniformization [5] is an often-used method for transforming a CTMC into a DTMC for transient analysis. Briefly, given a CTMC with finite state space $\mathcal{RS}$, generator matrix $\mathbf{Q}$, initial distribution vector $\mathbf{p}_0$, and reward vector $\mathbf{r}$, the uniformization procedure may be applied when the diagonal elements of $\mathbf{Q}$ are finite. Applying uniformization, we can compute $\mathbf{p}_s$, the transient distribution at time s, as

$$\mathbf{p}_s = \mathbf{p}_0 e^{(\mathbf{Q}s)} = \mathbf{p}_0 \sum_{k=0}^{\infty} \mathbf{P}^k \beta(\alpha s, k) \tag{2.1}$$

where $\alpha = \max_x(|Q(x,x)|)$, $\mathbf{P} = \mathbf{Q}/\alpha + \mathbf{I}$, and $\beta(\alpha s, k) = e^{-\alpha s}(\alpha s)^k/k!$ is the probability that a Poisson process with rate αs will make k jumps in the interval (0,1]. For a finite truncation of the infinite sum, lower and upper bounds for the probability $\mathbf{p}_s(x)$, $x \in \mathcal{RS}$, can be computed easily using standard means.

The instantaneous reward at time s is computed as

$$E[R_s] = \mathbf{p}_0 \left(\sum_{k=0}^{\infty} \mathbf{P}^k \beta(\alpha s, k) \right) \mathbf{r}. \tag{2.2}$$

Finite truncation of the sum can be bounded above and below by

$$\begin{aligned} &\mathbf{p}_0 \left(\sum_{k=0}^{K} \mathbf{P}^k \beta(\alpha s, k) \right) \mathbf{r} \;\leq\; E[R_s] \leq \\ &\mathbf{p}_0 \left(\sum_{k=0}^{K} \mathbf{P}^k \beta(\alpha s, k) \right) \mathbf{r} + \left(1 - \sum_{k=0}^{K} \beta(\alpha s, k) \right) \max_x(\mathbf{r}(x)). \end{aligned} \tag{2.3}$$

The expected value of the accumulated reward during the interval [0,s) is computed as

$$E[AR_s] = \int_0^s \mathbf{p}_s \mathbf{r} = \mathbf{p}_0 \left(\sum_{k=0}^{\infty} \mathbf{P}^k \frac{1 - \beta(\alpha s, k)}{\alpha} \right) \mathbf{r}, \tag{2.4}$$

and the bounds on the finite truncation of its sum can be computed in the same manner as

in (2.3).

2.4 Analyzable Class of Markovian Models and Their Underlying Matrix Structure

In this section, we describe a class of structured Markovian models that are analyzable by our approach. Many large, non-trivial models in reliability and performance analysis are built up from smaller logical components or submodels. The submodels are composed together by means of *state sharing* (e.g. [43]) or *action synchronization* (also known as *action sharing*). In our approach, we consider action synchronization. The structure in these models can be exploited to make their analysis more efficient. In particular, we consider the class of models for which we can represent generator matrices and solution vectors in a structured manner by means of Kronecker products and sums. In the following paragraphs, we describe the structure of the matrices underlying this class of models.

Suppose a given model has state space $\mathcal{RS}$ (where $|\mathcal{RS}| = n$) and is decomposable into J components, which are numbered from 1 to J. We usually use parenthesized superscripts and subscripts, i,$j \in \{1, \ldots, J\}$, to denote particular components, unless the context is clear (in which case, we do not use parentheses). $\mathcal{RS}_i = \{0, \ldots, n_i - 1\}$ is the set of states of component i, and the relation $\mathcal{RS} \subseteq \times_{i=1}^{J} \mathcal{RS}_i$ holds. Because the global state space is described by the states of the individual components, each global state can be represented by a J-dimensional vector $\mathbf{x}$ such that $\mathbf{x}(i)$ is the state of component i in global state $\mathbf{x}$. This J-dimensional representation can be linearized via $x = \sum_{i=1}^{J} \mathbf{x}(i) \cdot n_{i+1}^{J}$, where $n_j^i = \prod_{k=j}^{i} n_k$ and empty product equals 1. Both linearized and J-dimensional representations can be used interchangeably. This basic compositional representation of $\mathcal{RS}$ is the structure used in [26] and many other papers to represent $\mathbf{Q}$ in compact form. For numerical analysis, it is generally not sufficient to consider the cross product of component state spaces because of the unreachable states. However, for the bounding approach presented here, the representation is appropriate, because the path-based computation of rewards implicitly considers only reachable states. We will expand upon this feature further in Section 2.5, when we describe our path-based approach.

Components in a model interact via events or transitions from a set of synchronized transitions, $\mathcal{T}_S$. Furthermore, a component has local behavior described by local transitions. Let $\mathbf{Q}_l^{(i)}$ be the matrix of local transition rates of component i. $\mathbf{Q}_l^{(i)}$ is the generator matrix of a

CTMC with state space $\mathcal{RS}_i$. For each component i and synchronized transition $t \in \mathcal{T}_S$, let $\mathbf{E}_t^{(i)}$ be a matrix describing the transitions due to the occurrence of t in the state space of $\mathcal{RS}_i$. Thus, $\mathbf{E}_t^{(i)}(x, y)$ is the weight of the firing of t in local state x to transition to local state y in component i. If component i is not involved in the firing of t, then $\mathbf{E}_t^{(i)} = \mathbf{I}_{n_i}$, the identity matrix of order n_i. The global firing rate of transition $t \in \mathcal{T}_S$ starting in state $\mathbf{x}$ and ending in state $\mathbf{y}$ is given by $\lambda_t \prod_{i=1}^{J} \mathbf{E}_t^{(i)}(\mathbf{x}(i), \mathbf{y}(i))$, where λ_t is the rate of the transition. We assume that the weight of any transition $t \in \mathcal{T}_S$ is scaled such that $\max_{x \in \mathcal{RS}_i}(\sum_{y \in \mathcal{RS}_i} \mathbf{E}_t^{(i)}(x, y)) \leq 1.0$. In other words, matrices $\mathbf{E}_t^{(i)}$ have row sums between 0 and 1. This is to account for marking-dependent transition rates [30]; for constant transition rates, each row sum is either 0 or 1. The descriptor $\hat{\mathbf{Q}}$ of the CTMC, which contains the generator matrix $\mathbf{Q}$ as a submatrix, can be represented as the composition of component matrices:

$$\hat{\mathbf{Q}} = \bigoplus_{j=1}^{J} \mathbf{Q}_l^{(i)} + \sum_{t \in \mathcal{T}_S} \lambda_t \left(\bigotimes_{j=1}^{J} \mathbf{E}_t^{(i)} - \bigotimes_{j=1}^{J} \mathbf{D}_t^{(i)} \right), \tag{2.5}$$

where $\mathbf{D}_t^{(i)} = \mathrm{diag}(\mathbf{E}_t^{(i)} \mathbf{e}^T)$ and $\mathbf{e}$ is a n_i-dimensional vector of ones.

The representation of $\hat{\mathbf{Q}}$ in (2.5) is very compact, since all of the component matrices used in that equation have dimensions $n_i \times n_i$ rather than $n \times n$ (where n can be much larger than n_i). We assume that the initial distribution vector can also be composed from initial vectors of the components. Let $\mathbf{p}_0^{(i)}$ be the initial vector of component i ($\mathbf{p}_0^{(i)}$ can be obtained according to the decomposition of $\mathcal{RS}$ into $\mathcal{RS}_i$). The initial distribution of the complete model is a row vector that is given by $\mathbf{p}_0 = \bigotimes_{i=1}^{J} \mathbf{p}_0^{(i)}$ Similarly, let column vector $\mathbf{r}^{(i)}$ be the reward vector for component i; then $\mathbf{r} = \bigoplus_{i=1}^{J} \mathbf{r}^{(i)}$ or $\mathbf{r} = \bigotimes_{i=1}^{J} \mathbf{r}^{(i)}$ is the reward vector of the complete model. Whether the reward vector is built by the Kronecker sum or product depends on the type of reward. We can extend this beyond sums or products and use other associative and commutative operations like maximum or minimum.

Uniformization can be applied at the level of the components to yield a compact representation of matrix $\mathbf{P} = \mathbf{Q}/\Lambda + \mathbf{I}$, where $\Lambda = \sum_{i=1}^{J} \lambda_{l_i} + \sum_{t \in \mathcal{T}_S} \lambda_t$ and $\lambda_{l_i} = \max_{x \in \mathcal{RS}_i}(|\mathbf{Q}_l^{(i)}(x, x)|)$. Application of uniformization to a component matrix yields the transition matrix

$$\mathbf{P}_l^{(i)} = \mathbf{Q}_l^{(i)}/\lambda_{l_i} + \mathbf{I}, \tag{2.6}$$

which describes the transition probabilities of component i.

For each synchronized transition, $t \in \mathcal{T}_S$, we define matrices

$$\overline{\mathbf{E}}_t^{(i)} = \mathbf{I} - \mathbf{D}_t^{(i)}. \tag{2.7}$$

Matrix $\overline{\mathbf{E}}_t^{(i)}$ is a diagonal matrix that contains in position (x,x), $x \in \mathcal{RS}_i$, the probability that component i will not enable synchronized transition t in state x with respect to the events of a Poisson process with rate $\lambda_t s$. Due to the definition of the elements in $\mathbf{E}_t^{(i)}$, matrix $\mathbf{E}_t^{(i)} + \overline{\mathbf{E}}_t^{(i)}$ is a stochastic matrix. For each synchronized transition t, we have to distinguish between components that *may* disable the transition and those that *cannot* disable it. The latter include those components that do not synchronize at t. Thus, define $\mathcal{E}(t) = \{i | \exists 0 \leq x < n_i : \mathbf{e}_x \mathbf{E}_t^{(i)} \mathbf{e}^T < 1\}$ to be the set of components that may disable transition t. Set $\mathcal{E}(t)$ contains between 0 and J components. If all communication in the model is asynchronous, then $\mathcal{E}(t)$ contains at most one component. Furthermore, define set $\mathcal{S}(t)$, $t \in \mathcal{T}_S$, to contain all components i such that $\mathbf{E}_t^{(i)} \neq \mathbf{I}_{n_i}$. For local transition l_i, define $\mathcal{S}(l_i) = \{i\}$. In other words, the set $\mathcal{S}(t)$ contains all components (those that may and may not disable transition t) participating in transition t, whereas the set $\mathcal{E}(t)$ is a subset of $\mathcal{S}(t)$ and contains only those components that may disable t.

Matrix $\mathbf{P}$ then can be represented as

$$\begin{aligned}
\mathbf{P} = \Bigg(\sum_{i=1}^{J} \tfrac{\lambda_{l_i}}{\Lambda} \underbrace{\left(\bigotimes_{j=1}^{i-1} \mathbf{I}_{n_j} \right) \bigotimes \mathbf{P}_l^{(i)} \bigotimes \left(\bigotimes_{j=i+1}^{J} \mathbf{I}_{n_j} \right)}_{\mathbf{P}_{l_i}} \Bigg) + \\
\Bigg(\sum_{t \in \mathcal{T}_S} \tfrac{\lambda_t}{\Lambda} \underbrace{\left(\bigotimes_{i=1}^{J} \mathbf{E}_t^{(i)} \right)}_{\mathbf{P}_t} \Bigg) + \\
\Bigg(\sum_{t \in \mathcal{T}_S} \tfrac{\lambda_t}{\Lambda} \underbrace{\sum_{i \in \mathcal{E}(t)} \left(\bigotimes_{j=1}^{i-1} \mathbf{D}_t^{(j)} \right) \bigotimes \overline{\mathbf{E}}_t^{(i)} \bigotimes \left(\bigotimes_{j=i+1}^{J} \mathbf{I}_{n_j} \right)}_{\mathbf{P}_{\bar{t}_i}} \Bigg).
\end{aligned} \tag{2.8}$$

The first sum describes local transitions in the different components (matrices $\mathbf{P}_{l_i}$), and the remaining sums describe synchronized transitions (matrices $\mathbf{P}_t$ and $\mathbf{P}_{\bar{t}_i}$). The second sum describes *actual* synchronization among components in $\mathcal{S}(t)$ at synchronized transition t (matrix

$\mathbf{P}_t$). For each synchronized transition t, we have also the possibility that the transition may be disabled by at least one component. This *pseudo-synchronized* transition is described by the third sum. Each term in this sum (matrix $\mathbf{P}_{\bar{t}_i}$) describes the situation in which t is enabled by components with indices 1 through $(i-1)$, is disabled by component i, and may be enabled or disabled by components $(i+1)$ through J.

2.5 Structured Path-based Approach

In this section, we describe our path-based approach for computing transient rewards of large CTMCs. Often, approximate solutions are computed in these models, and thus it is also necessary to compute the upper and lower bounds on the solutions to determine whether the errors in the approximations are acceptable. This guarantee on the amount of error is especially relevant in many dependability analyses in which it is necessary to assure, for example, the minimal level of availability or reliability. We proceed by describing first the general concept of paths in our approach. Then we formalize the concept as an enumeration on strings and show how numerical computations can be done on paths. Because there may be many paths to enumerate, we also describe an equivalence relation among paths to reduce the number of paths that have to be explored and computed explicitly.

2.5.1 Conceptual Overview of the Approach

Conceptually, we consider paths at the higher level of local and synchronized transitions among components. By considering paths at this level, we gain several advantages. First, instead of considering each path individually, each step in our approach considers several paths at the same time. Thus, we can analyze a larger number of paths by computing a single vector-matrix product. Second, at the higher level of analysis, we can re-use previous computation. Third, by choosing the partition of the model and grouping of components appropriately, it is possible to choose some intermediate step between a complete numerical analysis and a path-based bound computation. The reward accumulated from analyzing a set of paths constitutes the lower bound for the reward of a CTMC. Knowing the probability of paths that have been considered, we can also compute the upper bound by assuming the maximum reward for all paths that have not been considered.

2.5.2 Numerical Analysis on Paths

The computation of path probabilities and rewards is efficient and simple in our approach because it requires only vector-matrix products of order n_i, rather than of order n. In our setting, paths are observable only by their transition labels, which correspond to different matrices. Let us define an alphabet $\mathcal{A} = \{l_1, \ldots, l_J\} \cup \mathcal{T}_S \cup (\cup_{t \in \mathcal{T}_S} \mathcal{E}(t))$, where again $l_i \in \{l_1, \ldots, l_J\}$ is a label signifying the occurrence of a local transition in component i. Let $\mathcal{P}$ be the set of all strings made from symbols in $\mathcal{A}$, and let $\mathcal{P}^l \subseteq \mathcal{P}$ be the set of strings of length l. For any $\boldsymbol{\pi} \in \mathcal{P}$, $\boldsymbol{\pi}(i) \in \mathcal{A}$ is the ith element and $|\boldsymbol{\pi}|$ is the length of path $\boldsymbol{\pi}$. Thus, $\mathcal{P}$ is the set of all paths of a model and $\mathcal{P}^l$ contains all paths of length l. Now consider a specific path $\boldsymbol{\pi} \in \mathcal{P}^l$, and let $\mathbf{p}_0$ be the initial distribution and $\mathbf{r}$ be the reward vector. The reward after the transition of the path can be computed by

$$R(\boldsymbol{\pi}) = \mathbf{p}_0 \left(\prod_{k=1}^{|\boldsymbol{\pi}|} \mathbf{P}_{\boldsymbol{\pi}(k)} \right) \mathbf{r} \ , \tag{2.9}$$

where

$$\begin{aligned}
\mathbf{P}_{l_i} &= \mathbf{I}_{n_1^{i-1}} \bigotimes \mathbf{P}_l^{(i)} \bigotimes \mathbf{I}_{n_{i+1}^J}, \\
\mathbf{P}_t &= \bigotimes_{i=1}^{J} \mathbf{E}_t^{(i)}, \text{ and} \\
\mathbf{P}_{\bar{t}_i} &= \bigotimes_{j=1}^{i-1} \mathbf{D}_t^{(j)} \bigotimes \overline{\mathbf{E}}_t^{(i)} \bigotimes \mathbf{I}_{n_{i+1}^J} \ .
\end{aligned}$$

The probability of the path $\boldsymbol{\pi}$ is given by

$$Prob(\boldsymbol{\pi}) = \prod_{k=1}^{|\boldsymbol{\pi}|} \frac{\lambda_{\boldsymbol{\pi}(k)}}{\Lambda} \tag{2.10}$$

where $\lambda_{\bar{t}_i} = \lambda_t$. The expected reward at time s and the expected accumulated reward in the interval $[0, s)$ can then be computed, respectively, in our path-based version by

$$E[R_s] = \sum_{l=0}^{\infty} \beta(\Lambda s, l) \sum_{\boldsymbol{\pi} \in \mathcal{P}^l} Prob(\boldsymbol{\pi}) R(\boldsymbol{\pi}) \tag{2.11}$$

$$E[AR_s] = \sum_{l=0}^{\infty} \frac{1}{\Lambda} \left(1 - \sum_{k=0}^{l} \beta(\Lambda s, k) \right) \sum_{\boldsymbol{\pi} \in \mathcal{P}^l} Prob(\boldsymbol{\pi}) R(\boldsymbol{\pi}) \tag{2.12}$$

The computation of path values can be done in an iterative manner. Define $\boldsymbol{\pi} \circ a$ as the path that results from $\boldsymbol{\pi}$ by appending $a \in \mathcal{A}$, and $a \circ \boldsymbol{\pi}$ as the path that results from $\boldsymbol{\pi}$ by prepending a. Let $\mathbf{p}[\boldsymbol{\pi}]$ be the state vector and $\mathbf{r}[\boldsymbol{\pi}]$ be the reward vector after $\boldsymbol{\pi}$ has been observed. Define $\mathbf{p}[\epsilon] = \mathbf{p}_0$ and $\mathbf{r}[\epsilon] = \mathbf{r}$, where ϵ is the empty string. Then vectors $\mathbf{p}[\boldsymbol{\pi}]$ and $\mathbf{r}[\boldsymbol{\pi}]$ can be computed iteratively:

$$\mathbf{p}[\boldsymbol{\pi} \circ a] = \mathbf{p}[\boldsymbol{\pi}]\mathbf{P}_a \text{ and } \mathbf{r}[a \circ \boldsymbol{\pi}] = \mathbf{P}_a \mathbf{r}[\boldsymbol{\pi}] \tag{2.13}$$

Theorem 1. *Vectors* $\mathbf{r}[\boldsymbol{\pi}]$ *and* $\mathbf{p}[\boldsymbol{\pi}]$ *can be represented in compact form as*

$$\mathbf{r}[\boldsymbol{\pi}] = \bigotimes_{i=1}^{J} \mathbf{r}^{(i)}[\boldsymbol{\pi}] \text{ and } \mathbf{p}[\boldsymbol{\pi}] = \bigotimes_{i=1}^{J} \mathbf{p}^{(i)}[\boldsymbol{\pi}]$$

where

$$\mathbf{r}^{(i)}[\boldsymbol{\pi}] = \prod_{k=1}^{|\boldsymbol{\pi}|} \boldsymbol{\Phi}^{(i)}_{\boldsymbol{\pi}(k)} \mathbf{r}^{(i)}, \; \mathbf{p}^{(i)}[\boldsymbol{\pi}] = \mathbf{p}_0^{(i)} \prod_{k=1}^{|\boldsymbol{\pi}|} \boldsymbol{\Phi}^{(i)}_{\boldsymbol{\pi}(k)}, \text{ and}$$

$$\boldsymbol{\Phi}^{(i)}_{\boldsymbol{\pi}(k)} = \begin{cases} \mathbf{P}_l^{(i)} & \text{if } \boldsymbol{\pi}(k) = l_i \\ \mathbf{E}_t^{(i)} & \text{if } \boldsymbol{\pi}(k) = t \text{ for } t \in \mathcal{T}_S \\ \mathbf{D}_t^{(i)} & \text{if } \boldsymbol{\pi}(k) = \bar{t}_j \text{ for } i < j \text{ and } t \in \mathcal{T}_S \\ \overline{\mathbf{E}}_t^{(i)} & \text{if } \boldsymbol{\pi}(k) = \bar{t}_i \text{ for } t \in \mathcal{T}_S \\ \mathbf{I}_{n_i} & \text{otherwise} \end{cases} .$$

Proof: We show the proof for $\mathbf{p}[\boldsymbol{\pi}]$ (the proof for $\mathbf{r}[\boldsymbol{\pi}]$ is completely analogous). Following (2.9), $\mathbf{p}[\boldsymbol{\pi}] = \mathbf{p}_0 \prod_{k=1}^{|\boldsymbol{\pi}|} \mathbf{P}_{\boldsymbol{\pi}(k)}$. Using basic properties of Kronecker products [26, 5] we obtain the following relation

$$\begin{aligned} \mathbf{p}[\boldsymbol{\pi}] &= \mathbf{p}_0 \prod_{k=1}^{|\boldsymbol{\pi}|} \mathbf{P}_{\boldsymbol{\pi}(k)} \\ &= \bigotimes_{i=1}^{J} \mathbf{p}_0^{(i)} \prod_{k=1}^{|\boldsymbol{\pi}|} \bigotimes_{i=1}^{J} \boldsymbol{\Phi}^{(i)}_{\boldsymbol{\pi}(k)} \\ &= \prod_{k=1}^{|\boldsymbol{\pi}|} \bigotimes_{i=1}^{J} \mathbf{p}_0^{(i)} \bigotimes_{i=1}^{J} \boldsymbol{\Phi}^{(i)}_{\boldsymbol{\pi}(k)} \\ &= \bigotimes_{i=1}^{J} \left(\prod_{k=1}^{|\boldsymbol{\pi}|} \mathbf{p}_0^{(i)} \boldsymbol{\Phi}^{(i)}_{\boldsymbol{\pi}(k)} \right) \end{aligned}$$

which proves the theorem for $\mathbf{p}[\boldsymbol{\pi}]$. ■

The above results show three important aspects of the method. First, values for paths can be computed efficiently, and the representation of the vectors remains compact. Second, state reachability is considered implicitly via uniformization in our formulation. Unreachable states and their paths have probabilities of zero. Hence, our approach does not have to incur the extra step of having to analyze reachability explicitly as is normally required in Kronecker approaches. Third, a path of length l is the originating path of $|\mathcal{A}|$ paths of length $l+1$. The set $\mathcal{P}^l$ contains $(|\mathcal{A}|)^l$ paths, which is an enormous number, even for small values of J and $|\mathcal{T}_S|$. Thus, it is crucial to be able to enumerate paths efficiently and to determine equivalent paths to avoid redundant computation. This issue will be addressed in the next subsection; for now, we consider how to compute bounds from sets of paths.

Without loss of generality, we assume that all rewards are non-negative[1]. In this case, each path contributes a non-negative amount to the reward solution, and any finite summation of path rewards yields a lower bound for the required result. Thus, if $\mathcal{AP}^l \subseteq \mathcal{P}^l$ is a subset of the set of paths of length l and $l_{max} = \max_l(\mathcal{AP}^l \neq \emptyset)$, then the bounds for $E[R_s]$ are

$$\begin{aligned}
&\sum_{l=0}^{l_{max}} \beta(\Lambda s, l) \sum_{\boldsymbol{\pi} \in \mathcal{AP}^l} Prob(\boldsymbol{\pi}) R(\boldsymbol{\pi}) \leq E[R_s] \leq \\
&\sum_{l=0}^{l_{max}} \beta(\Lambda s, l) \sum_{\boldsymbol{\pi} \in \mathcal{AP}^l} Prob(\boldsymbol{\pi}) R(\boldsymbol{\pi}) + \left(1 - \sum_{l=0}^{l_{max}} \beta(\Lambda s, l) \sum_{\boldsymbol{\pi} \in \mathcal{AP}^l} Prob(\boldsymbol{\pi}) \mathbf{p}[\boldsymbol{\pi}] \mathbf{e}^T \right) \max_x(\mathbf{r}(x)) \ .
\end{aligned} \tag{2.14}$$

We use a derivation from [44] to formulate a tight upper bound on $E[AR_s]$. Thus, the bounds for $E[AR_s]$ are

$$\begin{aligned}
&\sum_{l=0}^{l_{max}} \frac{1}{\Lambda} \left(1 - \sum_{k=0}^{l} \beta(\Lambda s, k) \right) \sum_{\boldsymbol{\pi} \in \mathcal{P}^l} Prob(\boldsymbol{\pi}) R(\boldsymbol{\pi}) \leq E[AR_s] \leq \\
&\sum_{l=0}^{l_{max}} \frac{1}{\Lambda} \left(1 - \sum_{k=0}^{l} \beta(\Lambda s, k) \right) \sum_{\boldsymbol{\pi} \in \mathcal{P}^l} Prob(\boldsymbol{\pi}) R(\boldsymbol{\pi}) \\
&\qquad + \left[\left(1 - \sum_{k=0}^{l} \beta(\Lambda s, k) \right) - \frac{l+1}{\Lambda t} \left(1 - \sum_{k=0}^{l+1} \beta(\Lambda s, k) \right) \right] \max_x(\mathbf{r}(x)) \cdot t \ .
\end{aligned} \tag{2.15}$$

[1]Negative rewards can be scaled to meet this condition.

2.5.3 Equivalence Relation Among Paths

Although our approach computes a whole set of paths for each path $\boldsymbol{\pi} \in \mathcal{P}$, the set of paths that need to be explored continues to grow exponentially. Thus, an efficient realization of path generation and analysis is very crucial. We consider now the equivalence among paths to help in reducing the number of paths that actually need to be analyzed. For $a, b \in \mathcal{A}$ and $\mathcal{S}(a) \cap \mathcal{S}(b) = \emptyset$, then $\mathbf{p}[\boldsymbol{\pi} \circ a \circ b] = \mathbf{p}[\boldsymbol{\pi} \circ b \circ a]$ and $\mathbf{r}[a \circ b \circ \boldsymbol{\pi}] = \mathbf{r}[b \circ a \circ \boldsymbol{\pi}]$. The reason is that matrix $\mathbf{P}_a$ modifies only the states of components in $\mathcal{S}(a)$ (and likewise, $\mathbf{P}_b$ modifies only those in $\mathcal{S}(b)$). This equivalence can be used to define an equivalence relation among paths. Let $\boldsymbol{\pi} \in \mathcal{P}^l$ and let $Perm(\boldsymbol{\pi})$ be a set of permutations on $\{1, \ldots, l\}$ such that

$$\begin{aligned} perm(x) = \;& y \Rightarrow \forall z \in \{\min(x,y), \ldots \max(x,y)\} \\ & : \; \mathcal{S}(\boldsymbol{\pi}(x)) \cap \mathcal{S}(\boldsymbol{\pi}(y)) \cap \mathcal{S}(\boldsymbol{\pi}(z)) = \emptyset \; . \end{aligned}$$

All paths that result from $\boldsymbol{\pi}$ by reordering of the elements according to some permutation of $Perm(\boldsymbol{\pi})$ are equivalent to $\boldsymbol{\pi}$ and have identical probabilities and rewards. The transitive closure of $Perm()$ defines an equivalence relation for the set of paths. Let $[\boldsymbol{\pi}]_\sim$ be the equivalence class that contains $\boldsymbol{\pi}$ and let $n_\sim(\boldsymbol{\pi})$ be the cardinality of the class. Then, after $\boldsymbol{\pi}$ has been explored, all paths from $[\boldsymbol{\pi}]_\sim$ can be considered to have been explored, and the rewards $R(\boldsymbol{\pi})$ and $AR(\boldsymbol{\pi})$ and probability $Prob(\boldsymbol{\pi})$ are multiplied by $n_\sim(\boldsymbol{\pi})$ for the computation of the lower and upper bounds.

2.6 Implementation Issues

Our approach is implemented as a C++ module that takes as input local and synchronized transition matrices and initial state probability and reward vectors. These matrices and vectors can be generated for a given model by existing tools such as the APNN Toolbox [45, 46] or a modified version of Möbius [47].

The current implementation of our approach explores and computes paths in a depth-first manner to minimize memory use in storing intermediate results. To further reduce memory demand, intermediate results are reused during path extension. These intermediate results include component state probability vectors and component reward vectors, which are much smaller than the complete model vectors. It should be emphasized that at no time during the computation are the complete model state probability and reward vectors computed or stored explicitly.

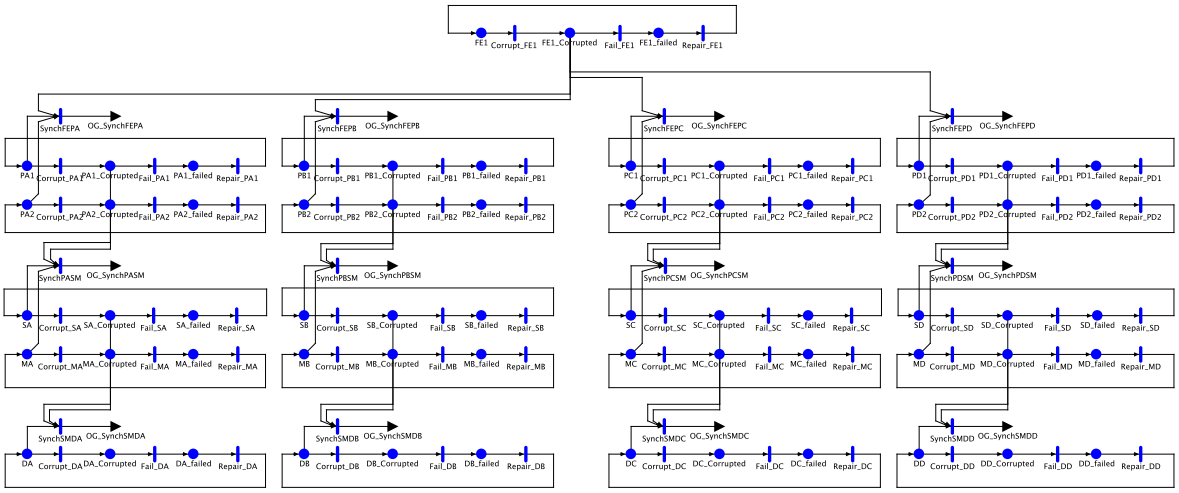


Figure 2.1: SAN model for computing the availability measure of a distributed information service system.

Our path generation algorithm generates only *canonical* paths. A canonical path is a representative path for a class of equivalent paths. Canonical paths have unique ordering of their nodes such that other canonical paths can be generated by considering only their last nodes. Equivalent paths are then calculated from the canonical paths, thus helping to improve performance and memory use by avoiding path lookups and storage. The current implementation exploits only the equivalence of paths from local transitions. Even so, the computational saving from exploitation of equivalent paths is quite substantial, as is shown in the next section.

2.7 Experimental Results

We evaluate our approach by studying its performance in analyzing a model of a distributed information service system adapted from the model in [48]. The example model describes the propagation of faults across the components constituting the system. After describing the system, we discuss the performance results we obtained from analyzing various configurations of the model. In this example, we show that the spread of the bounds shrinks rapidly with an increasing path length.

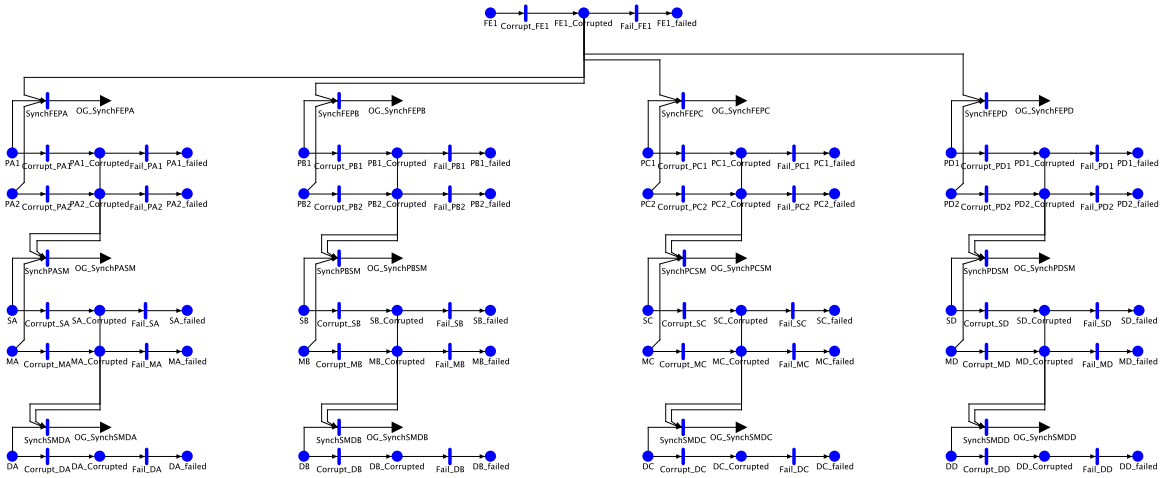


Figure 2.2: SAN model for computing the reliability measure of a distributed information service system.

2.7.1 Model Description

The information service system consists of a front-end module that interacts with four processing units. Each processing unit consists of two processors, one switch, one memory unit, and one database. Each of these components has its own repair facility. They all go through the cycle of *Working*, *Corrupted*, *Failed*, and *Repaired*. The *stochastic activity network* (SAN) model for computing the availability measure of the system is shown in Figure 2.1. Figure 2.2 shows the model for computing the reliability measure of the same system. Note the difference between the two models is that the former has repair facilities for each component of the system, whereas the latter does not.

Fault propagation in the system is modeled as follows:

- When the front-end is corrupted, it may propagate the error to any of the four processing units in which there are 2 working processors. Propagation occurs via the synchronized activities between the front-end and the processors in the processing units. The front-end or any of the processors may disable the synchronized activities. After propagating the error to a processing unit, the front-end may remain in the corrupted state and continue to propagate errors to other processing units until it fails or there are no more processing units to propagate the error.

- When both processors in a processing unit are corrupted, they both may propagate the

error to their working switch and memory via a synchronized activity. Any of the involved components may disable the synchronized activity. After the error propagation, the processors remain in the corrupted state until they fail.

- When both the switch and memory of a processing unit are corrupted, they may propagate their errors to the working database via a synchronized activity. Any of these components may disable the activity. After propagating the error, the switch and memory remain in the corrupted state until they fail.

We vary activity rates in the submodels and among the synchronized activities, so the resulting model does not have symmetries that would allow it to be lumped. In total, the model has 21 submodels and 12 synchronized activities. Because each submodel has three states, the state space of the whole model has $3^{21} \approx 10.5$ billion states.

In the following subsections, we discuss the performance results of our approach in computing reliability and availability for various configurations of the model. In particular, in all three configurations, we computed the reliability measure at transient time point 10.0 and the availability measure at transient time point 0.1 when all components in the model were in the working state. The two measures were computed at different time points because the high repair rates in the model would make computation of availability lengthy. Other variants of uniformization, such as adaptive uniformization [49], could be adapted to address this issue in a path-based approach. In addition to showing that our approach works for all configurations by converging to the simulated solution computed by Möbius, we also show how the sizes of the submodels in the configurations affect the performance of our approach. Due to space constraints, we show numerical data for all three configurations, but graphs for only the large-submodel-size configuration (more details are given in the following subsections). The trends in the graphs of the other two configurations are similar to that in the large-submodel-size configuration.

We evaluated all of our experiments on a workstation that had the AMD Athlon XP 2700+ processor running at 2.17 GHz with 512 MB of RAM. The operating system was Red Hat Linux 9.0 with mounted file systems. We compiled our implementation using the compiler g++ 3.3 with optimization flag -O3 only.

For all configurations, we observe that our implementation used a maximum of 1.32 MB of resident memory for data, code, and stack. In all, it used less than 5 MB of memory to include libraries. Tables 2.1 and 2.2 list the rates that we used for all of the activities in the model.

Table 2.1: Mean corruption, failure, and repair rates for the submodels

Submodels	Corruption Rate	Failure Rate	Repair Rate
Front-end	1/3000	10/1000	10/10
Processor A1	4/5000	4/1000	9/10
Processor B1	3/5000	4/1000	9/10
Processor C1	2/5000	4/1000	9/10
Processor D1	1/5000	4/1000	9/10
Processor A2	4/7000	4/1000	9/10
Processor B2	3/7000	4/1000	9/10
Processor C2	2/7000	4/1000	9/10
Processor D2	1/7000	4/1000	9/10
Switch A	4/11000	3/1000	8/10
Switch B	3/11000	3/1000	8/10
Switch C	2/11000	3/1000	8/10
Switch D	1/11000	3/1000	8/10
Memory A	4/13000	2/1000	7/10
Memory B	3/13000	2/1000	7/10
Memory C	2/13000	2/1000	7/10
Memory D	1/13000	2/1000	7/10
Database A	4/17000	1/1000	6/10
Database B	3/17000	1/1000	6/10
Database C	2/17000	1/1000	6/10
Database D	1/17000	1/1000	6/10

Table 2.2: Mean synchronized transition rates among submodels

Transition	Rate
SynchFEPA	4.5/3000
SynchFEPB	3.5/3000
SynchFEPC	2.5/3000
SynchFEPD	1.5/3000
SynchPASM	4.5/5000
SynchPBSM	3.5/5000
SynchPCSM	2.5/5000
SynchPDSM	1.5/5000
SynchSMDA	4.5/7000
SynchSMDB	3.5/7000
SynchSMDC	2.5/7000
SynchSMDD	1.5/7000

Table 2.3: Numerical results of reliability for the small-submodel-size configuration

Path Length	Lower E[R]	Upper E[R]	Lower E[AR]	Upper E[AR]	Canonical Paths	Potential Paths	Time (sec)
0	0.476569	1	7.06248	10	0	0	0.03
1	0.801045	0.97127	9.17246	9.81318	73	73	0
2	0.911506	0.950875	9.62039	9.73034	3620	5402	0.07
3	0.936575	0.94363	9.69364	9.70912	165671	394419	1.96
4	0.940843	0.941912	9.70335	9.70519	7.65514e+06	2.87927e+07	89.65
5	0.941424	0.941607	9.70443	9.70462	3.55539e+08	2.10186e+09	4177.58

Table 2.4: Numerical results of availability for the small-submodel-size configuration

Path Length	Lower E[R]	Upper E[R]	Lower E[AR]	Upper E[AR]	Canonical Paths	Potential Paths	Time (sec)
0	0.189985	1	0.0487723	0.1	0	0	0.03
1	0.505399	0.999885	0.0785352	0.0999892	73	73	0.05
2	0.767226	0.999695	0.0925224	0.099979	3620	5402	0.07
3	0.912122	0.999537	0.0977801	0.0999733	165671	394419	1.96
4	0.972261	0.99945	0.0994149	0.0999709	7.65514e+06	2.87927e+07	89.74
5	0.99223	0.999414	0.0998466	0.0999701	3.55539e+08	2.10186e+09	4145.85

2.7.2 Small-submodel-size Configuration

We begin with the configuration that has the smallest submodel size. In this configuration, each submodel represents a component, such as the front-end, processor, or database, as shown in Figures 2.1 and 2.1. Thus, each submodel has 3 states. Altogether there are 21 submodels and 12 synchronized activities in the configuration.

Tables 2.3 and 2.4 show the numerical results for the reliability and availability measures, respectively, computed for this configuration. Note the columns *Canonical Paths* and *Potential Paths*. The former shows the number of path sets (each set is represented by one canonical path) that our algorithm explored and computed; the latter shows the potential number of paths that would have to be explored and computed up to the corresponding path length if we had not used the path equivalence relation to identify equivalent paths. In general, the path equivalence relation can be better exploited when longer paths are considered. For this configuration and up to the path length of 5, the number of canonical paths computed to obtain the bounds was only 16.92% of the number of potential paths. Column *Time (sec)* shows the time taken to compute the paths.

The Möbius simulation results are $0.9413224 \pm 1.456671 \times 10^{-4}$ for the expected

Table 2.5: Numerical results of reliability for the medium-submodel-size configuration

Path Length	Lower E[R]	Upper E[R]	Lower E[AR]	Upper E[AR]	Canonical Paths	Potential Paths	Time (sec)
0	0.488145	1	7.13742	10	0	0	0.03
1	0.808787	0.970572	9.20375	9.81036	33	33	0.02
2	0.914095	0.950388	9.62794	9.72892	830	1122	0.02
3	0.937153	0.943459	9.69495	9.70873	19611	37059	0.2
4	0.940939	0.941872	9.70353	9.70512	465834	1.22298e+06	4.45
5	0.941437	0.941599	9.70445	9.70461	1.10816e+07	4.03584e+07	105.25
6	0.941491	0.941562	9.70454	9.70455	2.63546e+08	1.33183e+09	2502.38

instant-of-time reliability and $0.9993968 \pm 1.521793 \times 10^{-5}$ for the expected instant-of-time availability. The simulation results are $9.7036663 \pm 8.538210 \times 10^{-4}$ for the expected interval-of-time reliability and $0.0999397 \pm 8.711766 \times 10^{-7}$ for the expected interval-of-time availability. These results were obtained at a 95% confidence level.

Figures 2.3 and 2.4 show the graphs of the reliability and availability measures, respectively, for the large-submodel-size configuration. Similar trends are found for the small-submodel-size configuration. The only difference is the time needed for the bounds to converge toward the exact solution. A comparison of the convergence rates for the three configurations is shown in Figures 2.5 and 2.6 for the reliability and availability measures, respectively. Note that the larger configurations attain tighter bounds faster than the smaller configurations do.

2.7.3 Medium-submodel-size Configuration

For the medium-submodel-size configuration, we combined the two processors, the switch, and the memory in each processing unit into one submodel. There were four of these submodels in the complete model, and each of them had 81 states. In total, there were 9 submodels and 8 synchronized activities.

Tables 2.5 and 2.6 show the numerical results for the reliability and availability measures, respectively, computed for this configuration. Using the path equivalence relation, the number of canonical paths explored was only 19.79% of the potential number of paths.

Table 2.6: Numerical results of availability for the medium-submodel-size configuration

Path Length	Lower E[R]	Upper E[R]	Lower E[AR]	Upper E[AR]	Canonical Paths	Potential Paths	Time (sec)
0	0.19003	1	0.0487766	0.1	0	0	0.03
1	0.505475	0.999885	0.0785393	0.0999892	33	33	0.02
2	0.767289	0.999695	0.0925247	0.099979	830	1122	0.01
3	0.912156	0.999537	0.0977811	0.0999733	19611	37059	0.2
4	0.972275	0.99945	0.0994152	0.0999709	465834	1.22298e+06	4.46
5	0.992234	0.999414	0.0998467	0.0999701	1.10816e+07	4.03584e+07	105.21
6	0.997756	0.999402	0.0999456	0.0999699	2.63546e+08	1.33183e+09	2500.12

Table 2.7: Numerical results of reliability for the large-submodel-size configuration

Path Length	Lower E[R]	Upper E[R]	Lower E[AR]	Upper E[AR]	Canonical Paths	Potential Paths	Time (sec)
0	0.496585	1	7.19164	10	0	0	0.03
1	0.814259	0.970064	9.22574	9.80831	17	17	0.05
2	0.915869	0.950043	9.63308	9.72792	232	306	0.04
3	0.937536	0.943342	9.69582	9.70846	2871	5219	0.06
4	0.941001	0.941846	9.70364	9.70507	35437	88740	0.56
5	0.941445	0.941595	9.70446	9.7046	437967	1.5086e+06	6.54
6	0.941492	0.941561	9.70454	9.70455	5.41298e+06	2.56462e+07	81.2
7	0.941496	0.941557	9.70454	9.70454	6.68963e+07	4.35985e+08	997.02

2.7.4 Large-submodel-size Configuration

Lastly, for the large-submodel-size configuration, we combine the two processors, a switch, memory, and a database into one submodel. Of the three example configurations, this one has the largest submodels; each of them has 243 states. The complete model has 5 submodels and 4 synchronized activities.

Tables 2.7 and 2.8 show the numerical results for the reliability and availability measures, respectively, computed for this configuration. Using the path equivalence relation, the number of canonical paths explored to obtain the tight bounds shown in Figures 2.3 and 2.4 was only 15.34% of the number of potential paths. Note that because there are fewer submodels, our algorithm can extend the paths to longer lengths more quickly for this configuration than for others. As a result, tighter bounds can be obtained more quickly.

Figures 2.5 and 2.6 show the comparisons of the times needed to tighten the bounds around the solutions of the reliability and availability measures, respectively, for the three configurations.

Table 2.8: Numerical results of availability for the large-submodel-size configuration

Path Length	Lower E[R]	Upper E[R]	Lower E[AR]	Upper E[AR]	Canonical Paths	Potential Paths	Time (sec)
0	0.190063	1	0.0487796	0.1	0	0	0.05
1	0.505529	0.999885	0.0785422	0.0999892	17	17	0.05
2	0.767334	0.999695	0.0925263	0.099979	232	306	0.03
3	0.912181	0.999537	0.0977817	0.0999733	2871	5219	0.09
4	0.972286	0.99945	0.0994154	0.0999709	35437	88740	0.57
5	0.992238	0.999414	0.0998468	0.0999701	437967	1.5086e+06	6.59
6	0.997757	0.999402	0.0999456	0.0999699	5.41298e+06	2.56462e+07	81.63
7	0.999066	0.999398	0.0999656	0.0999699	6.68963e+07	4.35985e+08	1009.77

Except for the initial transient fluctuation[2] in the timing, which is due to the variability of time measurements below 0.1 seconds, the general trend is that the large-submodel-size configuration converges fastest among the three configurations.

2.8 Algorithmic Cost Analysis

We briefly compare the computational costs of the path-based approach against a standard numerical solution using uniformization. For the comparison, we make the following assumptions, which hold in many practical examples. The number of iterations of the uniformization approach is on the order of $O(\alpha s) \approx O(\Lambda s)$ assuming that $\Lambda \approx \alpha$. The number of states for each component is $O(n_1)$, and the number of reachable states is $O({n_1}^J)$. Matrices are sparse such that the effort for a vector-matrix product computation is $O(n_1)$ and $O({n_1}^J)$ for the path-based and complete approach, respectively. The cardinality of $\mathcal{A}$ equals L.

We begin with a comparison of the memory requirements and consider a depth-first enumeration of paths for the path-based approach. Furthermore, we assume that all intermediate results are stored, if they are required later. This means that up to $O(\Lambda s)$ intermediate results have to be stored and the storage requirements are in $O(n_1 \Lambda s)$. For a Kronecker-based representation of standard uniformization, storage requirements are thus $O(n_1^J)$. Therefore, the path-based approach requires less memory if $\Lambda s \leq n_1^{J-1}$. Since we are mainly interested in the analysis of large models that are partitioned into components of similar size, the path-based approach often requires less memory, even if all intermediate results are stored. Take, as an

[2]The erratic timing pattern is probably caused by the reading of data files (for the transition matrices and reward vectors) that resided on remote file systems.

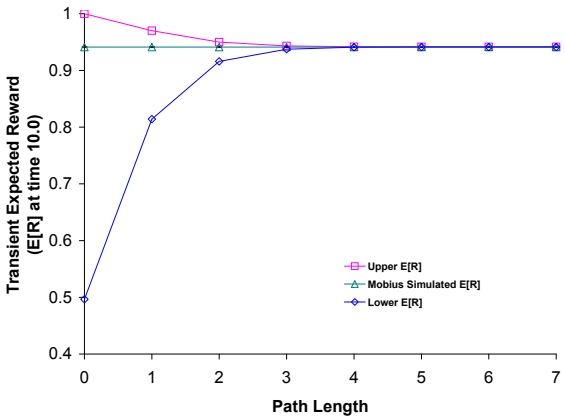
example, a system with 4 components and 10^2 states per component. Uniformization requires $O(10^8)$ memory, whereas the path-based approach requires $O(\Lambda s 10^2)$. For $\Lambda s < 10^6$, the path-based approach needs less memory. Furthermore, in uniformization each vector element has to be accessed in each iteration step, whereas intermediate results of the path-based approach need to be accessed less frequently, especially if they belong to short paths.

The computational effort of standard uniformization is $O(\Lambda s {n_1}^J)$ under the assumptions introduced above. The effort of the path-based approach depends on the number of paths to be explored. The number of paths depends on L and the sizes of the equivalence classes. We consider here two extreme cases. If no symbols can be exchanged, then the number of paths of length k equals L^k. If all symbols can be exchanged, then only $\binom{L+k-1}{L-1}$ different paths of length k have to be explored. If we consider the worst case, in which symbols cannot be exchanged, then the effort of the path-based approach is $O(n_1 L^{\Lambda s})$, whereas the effort of uniformization is $O(\Lambda s {n_1}^J)$. For $L > n_1$, the effort for the exhaustive path-based approach is similar to that for the uniformization approach if $\Lambda s \approx J$, which holds only for very short time horizons. If we consider the best case with independent transitions, then the path-based approach becomes more efficient, but it still outperforms uniformization only for small values of Λs.

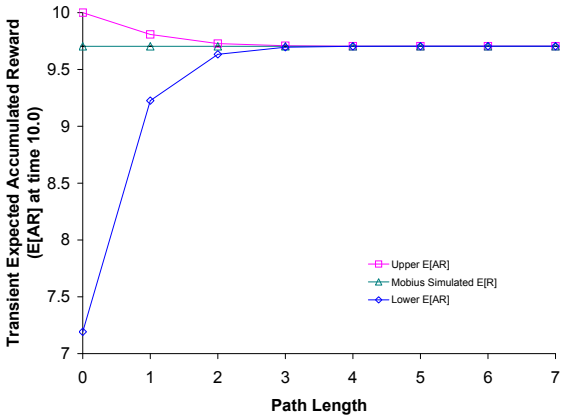
2.9 Summary

We presented a new approach that extends the current numerical capability to analyze very large Markov models for transient measures. We demonstrated the approach by analyzing a 10.5-billion-state model and showed that the computed bounds converged to results that were obtained through simulation.

In our approach, we extended existing path- and uniformization-based methods and formulated an equivalence relation among paths to identify sets of paths that are equivalent with respect to a reward measure. This relation allows us to explore multiple paths at the same time, thus significantly increasing the number of paths that need to be explored. Furthermore, the use of a structured representation for the state space and the direct computation of the desired reward measure (without ever storing the solution or iteration vectors) allow us to analyze very large models using a modest amount of storage. Additional benefits deriving from our approach include (1) implicit state reachability computation, (2) efficient path value computations via vector-matrix multiplications only, and (3) numerically stable computation.

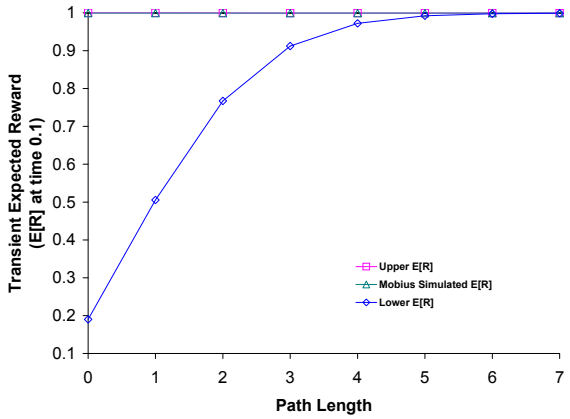


(a) Expected Reward

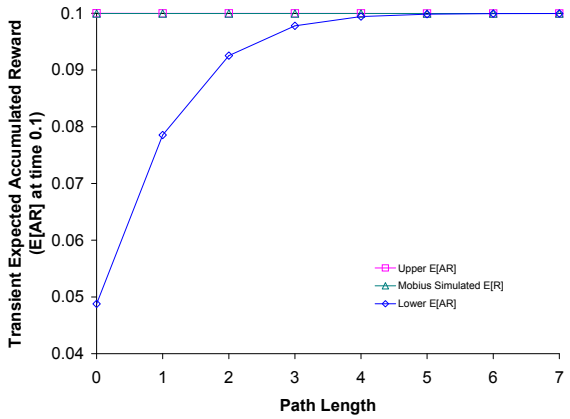


(b) Expected Accumulated Reward

Figure 2.3: Convergence of the bounds on the reliability measure toward the simulated solution obtained by Möbius for the large-submodel-size configuration.

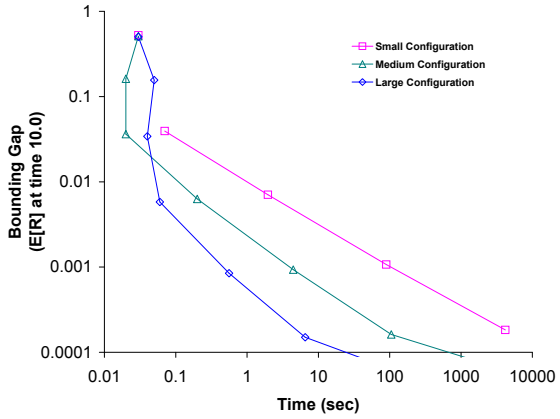


(a) Expected Reward

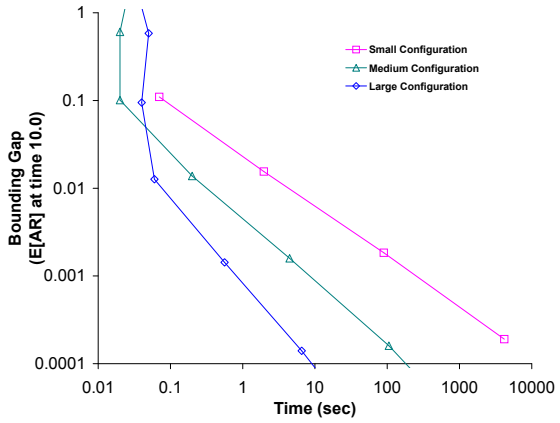


(b) Expected Accumulated Reward

Figure 2.4: Convergence of the bounds on the availability measure toward the simulated solution obtained by Möbius for the large-submodel-size configuration.

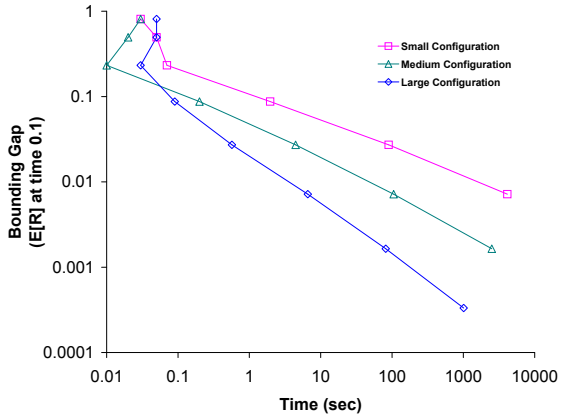


(a) Bounding Gap of Expected Reward

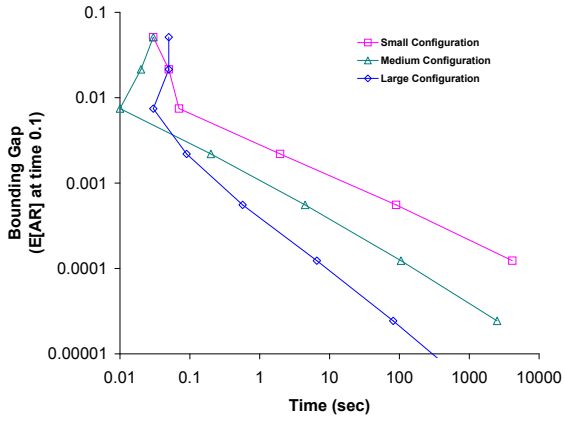


(b) Bounding Gap of Expected Accumulated Reward

Figure 2.5: Comparison of times to achieve tight bounds on the reliability measure.



(a) Bounding Gap of Expected Reward



(b) Bounding Gap of Expected Accumulated Reward

Figure 2.6: Comparison of times to achieve tight bounds on the availability measure.

CHAPTER 3

PATH-COMPOSITION APPROACH

3.1 Introduction

Due to ever-increasing size and complexity in system designs, model-based evaluation has become a cost-effective way to study alternative designs before the actual systems are built. Model-based evaluation can be used to estimate the reliability, availability, or performability of the systems, for instance. Often models are used that can be mapped onto continuous-time Markov chains (CTMCs) for solution. It is well-known that as models grow, the sizes of their state spaces grow at an exponential rate. That growth rate can quickly overwhelm the storage capacity of modern computing systems. Thus, new techniques that are effective at managing storage complexity are needed for analyzing large Markov models.

When transient results are required, several analysis methods are available. They can be roughly classified as either state-based or non-state-based and as either approximate or exact. Modern state-based techniques, such as [7, 8, 50, 51], are very effective in representing the state spaces and the corresponding transition matrices compactly, but they are limited by the fact that they must explicitly hold one or more solution or iteration vectors in memory. Since the sizes of those vectors are on the order of the state spaces, the techniques are restricted to solving small models. Consequently, non-state-based techniques like simulation have been generally used to solve large models. Simulation, however, belongs to the class of approximate techniques, since its results are statistical in nature.

An alternative to the previously mentioned techniques is the approach of path-based analysis (e.g., [36, 40, 1]), whereby bounded solutions are computed for large models. Like simulation, path-based techniques derive results for a model by evaluating trajectories over which the corresponding system may proceed over time. In contrast to the approach used in simulation, however, paths are selected in a systematic way. That usually allows one to compute upper and

lower bounds, instead of approximations, for the desired measures. Despite their success, existing path-based techniques find very limited application, because they suffer from poor performance. The performance of all path-based techniques is significantly dependent on two factors: (1) the number of paths they can compute in a given amount of time and (2) the relevance of the paths that are computed.

In this chapter, we introduce a novel technique for computing paths that improves performance substantially. The technique is based on the algorithm in which paths are composed from subpaths that are precomputed locally for the individual model components. The path composition algorithm helps to eliminate much of the redundant computation so that many more paths can be computed in a given amount of time. Furthermore, we augment the technique with a path-selection algorithm to choose the more relevant paths for analysis. To the best of our knowledge, this work is the first to propose the idea of path composition to improve the performance of path-based techniques for the analysis of Markov models. Our implementation of the algorithm effectively exploits the existence of common subpaths across all paths, resulting in a speedup of 6.6 to 8.8 times in our benchmark models.

The presentation of the materials in this chapter proceeds in the following order. In Section 3.2, we briefly review the necessary background material on the structured path-based approach. Here we focus specifically on the calculation of rewards and the algorithm for exploration at the path level. Then, in Section 3.3, we turn our attention to the subpath level and describe the new algorithm for exploring, computing, and composing subpaths. Section 3.4 presents experimental results for the new algorithm and compares them to those in our previous work. Finally, in Section 3.5, we conclude with a summary of our current work.

3.2 Review of the Structured Path-based Approach

In this section, we review the notations that are used, describe what paths are, and show how paths are explored and how reward is calculated on a path basis. A more detailed description and derivation can be found in [1]. This section lays the foundation for the derivation of the path-composition algorithm in the next section.

3.2.1 Computation of Path Reward and Solution Bounds

We consider a class of models whose infinitesimal generators and solution vectors can be composed from the component matrices and vectors using the Kronecker sum and product operators (see [1] for details). Specifically, the generator matrix $\mathbf{Q}$ (a submatrix of $\hat{\mathbf{Q}}$) of a model in the class can be represented by the composition of component matrices (which is typically used in the literature, e.g., see [5]):

$$\hat{\mathbf{Q}} = \bigoplus_{i=1}^{J} \mathbf{Q}_l^{(i)} + \sum_{t \in \mathcal{T}_S} \lambda_t \left(\bigotimes_{i=1}^{J} \mathbf{E}_t^{(i)} - \bigotimes_{i=1}^{J} \mathbf{D}_t^{(i)} \right), \tag{3.1}$$

where $\mathbf{Q}_l^{(i)}$ is the generator matrix of local transitions in component i ($1 \leq i \leq J$), $\mathcal{T}_S$ is the set of synchronized transitions, λ_t is the rate of synchronized transition t, $\mathbf{E}_t^{(i)}$ is the synchronized transition matrix with respect to component i and transition t, $\mathbf{D}_t^{(i)} = \text{diag}(\mathbf{E}_t^{(i)}\mathbf{e}^T)$, and $\mathbf{e}$ is an n_i-dimensional vector of ones.

Let $\mathcal{RS}_i = \{0, \ldots, n_i - 1\}$ be the set of states of component i, and define $\lambda_{l_i} = \max_{x \in \mathcal{RS}_i}(|\mathbf{Q}_l^{(i)}(x,x)|)$ and $\Lambda = \sum_{i=1}^{J} \lambda_{l_i} + \sum_{t \in \mathcal{T}_S} \lambda_t$. Applying uniformization to the generator matrix of component i yields the transition matrix,

$$\mathbf{P}_l^{(i)} = \mathbf{Q}_l^{(i)}/\lambda_{l_i} + \mathbf{I} \tag{3.2}$$

for a DTMC embedded in a Poisson process of rate $\lambda_{l_i} s$ that has density

$$\beta(\lambda_{l_i} s, k) = e^{-\lambda_{l_i} s} \cdot \frac{(\lambda_{l_i} s)^k}{k!}.$$

Further, define $\mathcal{E}(t) = \{i \mid \exists\, 0 \leq x < n_i : \mathbf{e}_x \mathbf{E}_t^{(i)} \mathbf{e}^T < 1\}$ to be the set of components that may disable transition t. For each synchronized transition $t \in \mathcal{T}_S$, we define the following matrices, which correspond to elements in the set $\mathcal{E}(t)$:

$$\overline{\mathbf{E}}_t^{(i)} = \mathbf{I} - \mathbf{D}_t^{(i)}. \tag{3.3}$$

A path in the approach consists of a sequence of component-level transitions. The generation of paths can be considered as an enumeration of strings from the alphabet $\mathcal{A} = \{l_1, \ldots, l_J\} \cup \mathcal{T}_S \cup (\cup_{t \in \mathcal{T}_S} \mathcal{E}'(t))$, where l_i denotes a local transition in component i and

$\mathcal{E}'(t) = \{\bar{t}_i \mid i \in \mathcal{E}(t)\}$ is the set of artificial transitions of which $\bar{t}_i$ for component i may be generated if the rate of synchronized transition t depends on the states of i. Let $\mathcal{P}$ be the language of all strings over $\mathcal{A}$, and let $\mathcal{P}^l \subseteq \mathcal{P}$ be the language of strings of length l. For any $\boldsymbol{\pi} \in \mathcal{P}$, $\boldsymbol{\pi}(i) \in \mathcal{A}$ is the ith element and $|\boldsymbol{\pi}|$ is the length of the string $\boldsymbol{\pi}$. Thus, $\mathcal{P}$ corresponds to the set of all paths over which a model may evolve, and $\mathcal{P}^l$ contains all paths of length l. Now consider a specific path $\boldsymbol{\pi} \in \mathcal{P}^l$, and let $\mathbf{p}_0 = \otimes_{j=1}^{J} \mathbf{p}_0^{(j)}$ be the initial state distribution vector and $\mathbf{r} = \odot_{j=1}^{J} \mathbf{r}^{(j)}$ be the reward vector. Analogously, let $\mathbf{p}_0^{(i)}$ and $\mathbf{r}^{(i)}$ be the initial distribution and reward vectors for component i, and let $\odot$ denote either $\otimes$ or $\oplus$ based on whether reward is computed by a product or a sum of constituent component rewards. The reward after the transition of the path is

$$R(\boldsymbol{\pi}) = \bigotimes_{j=1}^{J} \mathbf{p}_0^{(j)} \left(\prod_{k=1}^{|\boldsymbol{\pi}|} \mathbf{P}_{\boldsymbol{\pi}(k)} \right) \bigodot_{j=1}^{J} \mathbf{r}^{(j)} \ , \tag{3.4}$$

where, for $n_j^i = \prod_{k=j}^{i} n_k$,

$$\begin{aligned}
\mathbf{P}_{l_i} &= \mathbf{I}_{n_1^{i-1}} \bigotimes \mathbf{P}_l^{(i)} \bigotimes \mathbf{I}_{n_{i+1}^J}, \\
\mathbf{P}_t &= \bigotimes_{i=1}^{J} \mathbf{E}_t^{(i)}, \text{ and} \\
\mathbf{P}_{\bar{t}_i} &= \bigotimes_{j=1}^{i-1} \mathbf{D}_t^{(j)} \bigotimes \overline{\mathbf{E}}_t^{(i)} \bigotimes \mathbf{I}_{n_{i+1}^J} \ .
\end{aligned}$$

Note that each path at the component level (as shown above) corresponds to a set of trajectories at the state level. Further, observe that there are countably infinitely many paths over which a model may evolve. Thus, although a set of trajectories can be computed simultaneously, evaluating a large number of paths efficiently is still an important performance consideration. The probability of the path $\boldsymbol{\pi}$ is given by $Prob(\boldsymbol{\pi}) = \prod_{k=1}^{|\boldsymbol{\pi}|} \frac{\lambda_{\boldsymbol{\pi}(k)}}{\Lambda}$, where $\lambda_{\bar{t}_i} = \lambda_t$. The expected reward at time s and the expected accumulated reward in the interval $[0, s)$ can then be computed by

$$E[R_s] = \sum_{l=0}^{\infty} \beta(\Lambda s, l) \sum_{\boldsymbol{\pi} \in \mathcal{P}^l} Prob(\boldsymbol{\pi}) R(\boldsymbol{\pi}) \tag{3.5}$$

$$E[AR_s] = \sum_{l=0}^{\infty} \frac{1}{\Lambda} \left(1 - \sum_{k=0}^{l} \beta(\Lambda s, k) \right) \sum_{\boldsymbol{\pi} \in \mathcal{P}^l} Prob(\boldsymbol{\pi}) R(\boldsymbol{\pi}). \tag{3.6}$$

Equations (3.5) and (3.6) state that the expected rewards are just weighted sums of path rewards

$R(\boldsymbol{\pi})$ for all paths $\boldsymbol{\pi} \in \mathcal{P}$. The lower and upper bounds on the finite truncations of (3.5) and (3.6) can be computed according to the derivation in [1].

3.2.2 Algorithm for Exploring Paths

As shown in the previous section, a model may evolve over many paths. Exploring and computing those paths efficiently is a critical consideration in achieving good performance. Since the structured path-based approach is formulated at a higher level of path abstraction, we can use a partial order reduction method to reduce considerably the number of paths that the approach has to explore directly. In particular, the approach explores only *canonical paths*. Each canonical path is a representative path of a set of equivalent paths induced by a partial order reduction relation. Once a canonical path has been explored, all of the equivalent paths can be determined from it inexpensively (see [1] for details on the path equivalence relation). In addition to exploring only canonical paths, an efficient algorithm ought to reuse as many results from the computed canonical paths as possible. The algorithms for exploring canonical paths and efficiently computing equivalent paths are described below.

Canonical paths can be explored through a definition of a lexicographical order among them. That is done through the imposition of an order among the transitions in $\mathcal{A} = \{l_1, \ldots, l_J\} \cup \mathcal{T}_S \cup (\cup_{t \in \mathcal{T}_S} \mathcal{E}'(t))$. Given two paths, $\boldsymbol{\pi}_1 = \boldsymbol{\pi} \circ a$ and $\boldsymbol{\pi}_2 = \boldsymbol{\pi} \circ b$, $a, b \in \mathcal{A}$, $\boldsymbol{\pi} \in \mathcal{P}$, then $\boldsymbol{\pi}_2$ is lexicographically greater than $\boldsymbol{\pi}_1$ if b is lexicographically greater than a according to the ordering of the transitions in $\mathcal{A}$. Thus, an algorithm for exploring canonical paths considers only the lexicographically largest path among a set of equivalent paths.

More formally, we explore canonical paths in the following way. Let $\mathcal{TS} = \mathcal{T}_S \cup (\cup_{t \in \mathcal{T}_S} \mathcal{E}'(t))$ be the set of all synchronized transitions, ε be the null path, $\mathcal{CP}$ be the set of canonical paths, and $\mathcal{CP}^l$ be the set of canonical paths of length l. Then, $\mathcal{CP}^0 = \{\varepsilon\}$ and $\mathcal{CP} = \bigcup_{l \geq 0} \mathcal{CP}^l$. For $l > 0$, the set $\mathcal{CP}^l$ can be generated lexicographically in accordance with

$$\begin{aligned} \mathcal{CP}^l = \Big\{ & \boldsymbol{\pi} \circ a \mid \boldsymbol{\pi} \in \mathcal{CP}^{l-1},\ a \in \mathcal{TS} \vee \\ & \big(a = l_i \wedge \big(\boldsymbol{\pi}(|\boldsymbol{\pi}|) \in \mathcal{TS} \vee \big(\boldsymbol{\pi}(|\boldsymbol{\pi}|) = l_j \wedge j \leq i\big)\big)\big) \Big\}. \end{aligned} \tag{3.7}$$

Using this canonical-path exploration algorithm, we can reuse the computed results (such as the state distribution vectors) from paths that have already been explored. The algorithm uses a depth-first exploration strategy to minimize the amount of required storage by eliminating the

need to store all intermediate results from the computed canonical paths.

Let $card(\boldsymbol{\pi})$ be the cardinality of the set of paths equivalent to $\boldsymbol{\pi}$. Each path $\boldsymbol{\pi} \in \mathcal{CP}$ represents $card(\boldsymbol{\pi})$ paths in $\mathcal{P}$ whose rewards, probabilities, and state distribution vectors are all identical. For the computation of $card(\boldsymbol{\pi})$, we define the functions $cs(\boldsymbol{\pi})$ and $cl^{(i)}(\boldsymbol{\pi})$, $i = 1, \ldots, J$, on paths in the following way. Let $cs(\varepsilon) = cl^{(i)}(\varepsilon) = 0$ and

$$cs(\boldsymbol{\pi} \circ a) = \begin{cases} card(\boldsymbol{\pi}) & \text{if } a \in \mathcal{TS} \vee \\ & (a \notin \mathcal{TS} \wedge \boldsymbol{\pi}(|\boldsymbol{\pi}|) \in \mathcal{TS}), \\ cs(\boldsymbol{\pi}) & \text{otherwise} \end{cases} \tag{3.8}$$

$$cl^{(i)}(\boldsymbol{\pi} \circ a) = \begin{cases} cl^{(i)}(\boldsymbol{\pi}) + 1 & \text{if } a = l_i \\ 0 & \text{if } a \in \mathcal{TS}. \\ cl^{(i)}(\boldsymbol{\pi}) & \text{otherwise} \end{cases} \tag{3.9}$$

The function $cs(\boldsymbol{\pi})$ computes the cardinality up to the last synchronized transition in $\boldsymbol{\pi}$, and $cl^{(i)}(\boldsymbol{\pi})$ counts the local transitions after the synchronized transition. Both functions together implement a partial order reduction to compute equivalent paths in which the local transitions between two immediate synchronized transitions can be arbitrarily interchanged. The algorithm uses those functions to compute $card(\boldsymbol{\pi})$ in the following way:

$$card(\boldsymbol{\pi}) = cs(\boldsymbol{\pi}) \cdot \frac{\left(\sum_{i=1}^{J} cl^{(i)}(\boldsymbol{\pi})\right)!}{\prod_{j=1}^{J} \left(cl^{(i)}(\boldsymbol{\pi})!\right)} \tag{3.10}$$

In this section, we described briefly how a model is solved through the computation of various attribute values at the path level. In the next section, we show how the new technique of path composition can compute those values more efficiently at the subpath level.

3.3 Algorithms for Path Composition and Path Selection

We now present the main technical contribution of this chapter. This section is presented in the following manner. First, we give the justification for how paths can be composed from subpaths. This is done by showing how paths can be decomposed into subpaths. Next, we show how the subpath values are computed and, subsequently, how they can be used to compute the path

values. We then present the complete and necessarily efficient algorithm for composing the subpaths. The last part of this section describes an approach for selecting important paths from the precomputed subpaths.

3.3.1 The Path-Composition Algorithm

The essence of the path-composition algorithm is that it is much quicker to compute paths by composing subpaths than to compute the paths themselves directly as in (3.4), (3.5), and (3.6). To that end, we explain here how they can be composed by showing how they can be decomposed into subpaths. Let l_1 and l_2 be the local transitions from components 1 and 2, $t_{\{1,2\}}$ be a synchronized transition between components 1 and 2, and $t_{\{4,5\}}$ be a synchronized transition that does not affect components 1 and 2. There are three general cases to consider:

Case 1: Given that a path consists of local transitions only, such as $l_1 \circ l_2 \circ l_1 \circ l_2$, we can decompose it into two subpaths $l_1 \circ l_1$ and $l_2 \circ l_2$, since l_1 does not affect component 2 and similarly l_2 does not affect component 1.

Case 2: Given that a path consists of affecting synchronized transitions, such as $l_1 \circ l_2 \circ t_{\{1,2\}} \circ l_1 \circ l_2$, we can decompose it into two subpaths $l_1 \circ t_{\{1,2\}} \circ l_1$ and $l_2 \circ t_{\{1,2\}} \circ l_2$, in which we compute the effect of the synchronized transition on the components individually.

Case 3: Given that a path consists of non-affecting synchronized transitions, such as $l_1 \circ l_2 \circ t_{\{4,5\}} \circ l_1 \circ l_2$, we can decompose it into two subpaths $l_1 \circ l_1$ and $l_2 \circ l_2$, since the synchronized transition has no effect on the components.

Thus, paths can be composed through identification of the sets of subpaths that make up the paths. Moreover, the subpaths can be computed individually and, thereafter, characterized by a few real values. We present the path-composition algorithm in detail in the next subsection. In the rest of this subsection, we show how the subpath values are computed.

Let us assume for the moment that all components in a model participate in all synchronizations. That assumption can be relaxed easily later without affecting the correctness of the analysis; doing so now would unduly complicate the discussion and the notations used. Define $\mathcal{A}^{(i)} = \mathcal{T}_S \bigcup (\bigcup_{t \in \mathcal{T}_S} \bar{t}_i) \bigcup \{l_i\}$ to be the set of all transitions that may affect component i. For a given path $\boldsymbol{\pi}$, let $\boldsymbol{\pi}^{(i)}$ be a basic subpath of $\boldsymbol{\pi}$ consisting only of transitions in $\mathcal{A}^{(i)}$. Thus, $\boldsymbol{\pi}^{(i)}$ is the projection of $\boldsymbol{\pi}$ onto $\mathcal{A}^{(i)}$.

As shown in [1], the state distribution at the end of a path is computed by the Kronecker product of the component vectors. As a result, we can compute the state distribution vector for component i after the transition of the subpath $\boldsymbol{\pi}^{(i)}$ by

$$\mathbf{p}^{(i)}[\boldsymbol{\pi}^{(i)}] = \mathbf{p}_0^{(i)} \prod_{k=1}^{\left|\boldsymbol{\pi}^{(i)}\right|} \boldsymbol{\Phi}_{\boldsymbol{\pi}^{(i)}(k)}, \tag{3.11}$$

where

$$\boldsymbol{\Phi}_{\boldsymbol{\pi}^{(i)}(k)} = \begin{cases} \mathbf{P}_l^{(i)} & \text{if } \boldsymbol{\pi}^{(i)}(k) = l_i \\ \mathbf{E}_t^{(i)} & \text{if } \boldsymbol{\pi}^{(i)}(k) = t \text{ for } t \in \mathcal{T}_S \\ \mathbf{D}_t^{(i)} & \text{if } \boldsymbol{\pi}^{(i)}(k) = \bar{t}_j \text{ for } i < j \text{ and } t \in \mathcal{T}_S \\ \overline{\mathbf{E}}_t^{(i)} & \text{if } \boldsymbol{\pi}^{(i)}(k) = \bar{t}_i \text{ for } t \in \mathcal{T}_S \\ \mathbf{I}_{n_i} & \text{otherwise} \end{cases}.$$

The state distribution vector after the transition of the path $\boldsymbol{\pi}$ is $\mathbf{p}[\boldsymbol{\pi}] = \otimes_{j=1}^{J} \mathbf{p}^{(j)}[\boldsymbol{\pi}^{(j)}]$, and from (3.4), the reward of the path is computed by

$$R[\boldsymbol{\pi}] = \left(\bigotimes_{j=1}^{J} \mathbf{p}^{(j)}[\boldsymbol{\pi}^{(j)}] \right) \left(\bigodot_{j=1}^{J} \mathbf{r}^{(j)} \right).$$

Note that $\mathbf{p}^{(j)}[\boldsymbol{\pi}^{(j)}]$ is a row vector, whereas $\mathbf{r}^{(i)}$ is a column vector. If $\odot$ is $\otimes$ (i.e., rewards are computed as a product of component rewards), then

$$\begin{aligned} R[\boldsymbol{\pi}] &= \bigotimes_{j=1}^{J} \mathbf{p}^{(j)}[\boldsymbol{\pi}^{(j)}] \bigotimes_{j=1}^{J} \mathbf{r}^{(j)} \\ &= \bigotimes_{j=1}^{J} \mathbf{p}^{(j)}[\boldsymbol{\pi}^{(j)}] \mathbf{r}^{(j)} \\ &= \prod_{j=1}^{J} \mathbf{p}^{(j)}[\boldsymbol{\pi}^{(j)}] \mathbf{r}^{(j)}. \end{aligned}$$

When $\odot$ is $\oplus$, the computation is slightly more complex, yielding

$$\begin{aligned} R[\boldsymbol{\pi}] = & \left(\bigotimes_{j=1}^{J} \mathbf{p}^{(j)}[\boldsymbol{\pi}^{(j)}] \right) \left(\bigoplus_{j=1}^{J} \mathbf{r}^{(j)} \right) \\ = & \sum_{j=1}^{J} \left(\bigotimes_{i=1}^{J} \mathbf{p}^{(i)}[\boldsymbol{\pi}^{(\boldsymbol{i})}] \right) \left(\mathbf{e}_{n_1^{i-1}}^{T} \bigotimes \mathbf{r}^{(j)} \bigotimes \mathbf{e}_{n_{i+1}^{J}}^{T} \right) \\ = & \sum_{j=1}^{J} \left(\mathbf{p}^{(j)}[\boldsymbol{\pi}^{(j)}] \mathbf{r}^{(j)} \left(\prod_{i=1, i \neq j}^{J} \mathbf{p}^{(i)}[\boldsymbol{\pi}^{(\boldsymbol{i})}] \mathbf{e}_{n_i}^{T} \right) \right) \end{aligned}$$

where $\mathbf{e}_n$ is a row vector of length n with all elements equal to 1. For notational convenience define

$$\nu^{(i)}(\boldsymbol{\pi}^{(\boldsymbol{i})}) = \mathbf{p}^{(i)}[\boldsymbol{\pi}^{(\boldsymbol{i})}] \mathbf{r}^{(i)}$$

and

$$\xi^{(i)}(\boldsymbol{\pi}^{(\boldsymbol{i})}) = \prod_{j=1, i \neq j}^{J} \mathbf{p}^{(j)}[\pi^{(j)}] \mathbf{e}_{n_j}^{T}$$

such that the reward for the path $\boldsymbol{\pi}$ can be computed as either

$$R[\boldsymbol{\pi}] = \prod_{i=1}^{J} \nu^{(i)}(\boldsymbol{\pi}^{(\boldsymbol{i})}) \quad \text{or} \quad \sum_{i=1}^{J} \nu^{(i)}(\boldsymbol{\pi}^{(\boldsymbol{i})}) \cdot \xi^{(i)}(\boldsymbol{\pi}^{(\boldsymbol{i})}) \ . \tag{3.12}$$

Similarly, the probability of a path can be decomposed into the subpath probabilities due to the local and synchronized transitions. The probability of the subpath due to the local transitions is

$$ProbL(\boldsymbol{\pi}^{(\boldsymbol{i})}) = \prod_{k \in \{1, \ldots, \left| \boldsymbol{\pi}^{(\boldsymbol{i})} \right| \} \wedge \boldsymbol{\pi}^{(\boldsymbol{i})}(k) = l_i} \frac{\lambda_{l_i}}{\Lambda},$$

and the probability due to the synchronized transitions is

$$ProbS(\boldsymbol{\pi}) = \prod_{k \in \{1, \ldots, |\boldsymbol{\pi}|\} \wedge \boldsymbol{\pi}(k) \in \mathcal{TS}} \frac{\lambda_{\boldsymbol{\pi}(k)}}{\Lambda},$$

such that

$$Prob(\boldsymbol{\pi}) = ProbS(\boldsymbol{\pi}) \prod_{i=1}^{J} ProbL(\boldsymbol{\pi}^{(i)}). \tag{3.13}$$

Thus, we can exploit redundant computation across many paths to improve performance by first decomposing the paths into basic subpaths. Various real values of the subpaths are then precomputed and cached. Afterward, the subpaths can be composed, and their cached values can be used to compute the bounds on the rewards computed by (3.5) and (3.6).

Before we introduce the algorithm for composing the subpaths, we present a small example to show the advantages of an isolated computation and caching of $\nu^{(i)}(\boldsymbol{\pi^{(i)}})$ and $\xi^{(i)}(\boldsymbol{\pi^{(i)}})$. A model that has two components and a single synchronized transition t that can be disabled only by the first component will be analyzed. Thus, given $\mathcal{A} = \{l_1, l_2, t, \bar{t}_1\}$, we consider the set of paths of length 3, and we assume that each matrix $\mathbf{P}_l^{(i)}$ contains nz non-zeros and that the remaining matrices contain $nz/2$ elements such that the effort it takes to analyze a path of length x equals $x \cdot nz$. $\mathcal{P}^3$ contains $4^3 = 64$ different paths, and since each path requires a computational effort of $3nz$, the overall cost for this naive computation is on the order of $192nz$.

Observe that the number of paths at the state level is usually much larger and that every path in our approach usually represents a large number of paths at the state level. By storing intermediate vectors $\mathbf{p}^{(i)}[\boldsymbol{\pi^{(i)}}]$, one can avoid having to recompute the vectors many times. Thus, if path $l_1 l_1 l_1$ has been analyzed, path $l_1 l_1 l_2$ can be computed with a cost of only nz. From a computational point of view, the computation requires a depth-first traversal of the tree that describes all possible paths. The number of vectors to be stored equals the depth of the tree, which is acceptable since the vectors have dimensions that correspond to the sizes of the component state spaces and not the overall state space. Because already-computed vectors are reused, the number of required operations is proportional to $81nz$. Exploitation of the partial order reduction reduces the number of paths from 64 to 52. The effect of the reduction is relatively small for this example, since the path length is short and we consider only two components. Thus, only the sequences of local transitions can be reordered; for example, $l_1 l_1 l_2$, $l_1 l_2 l_1$, and $l_2 l_1 l_1$ are equivalent paths, and the first is the canonical path. If intermediate results are reused, the cost of the path computation using the canonical paths is proportional to $76nz$.

If we consider local paths for the components, then the first component describes $3^3 = 27$ paths. For the estimation of the computational cost, recall that at the component level, the multiplication of a vector with a matrix describing a synchronization requires an effort in $0.5nz$ rather than nz. Since there is 1 path with 3 local transitions, there are 6 paths with 2 local transitions, there are 12 paths with 1 local transition, and there are 8 paths without local

transitions, the total cost is proportional to $54nz$. Additionally, since vectors at the component level can be reused, the cost can be reduced to $26nz$. For the second component, we have only two different transitions, because the synchronized transition can be disabled only by the first component. Thus, there are 8 different paths, and the computational cost is proportional to $18nz$ without reuse of intermediate vectors and is proportional to $10.5nz$ with reuse of the vectors. If we reuse vectors, then the overall cost is proportional to $36.5nz$. Afterward, all values $\nu^{(i)}(\boldsymbol{\pi}^{(i)})$ and $\xi^{(i)}(\boldsymbol{\pi}^{(i)})$ are known for local paths with lengths up to 3. The values can be used in (3.12) for the computation of the reward values.

Observe that using the local paths with lengths up to 3, it is possible to compute rewards for all global paths with lengths less than or equal to 3 and also some other global paths with lengths 4, 5, and 6. Computing those additional global paths of lengths greater than 3 is inexpensive, and it helps to tighten the lower bounds further. To provide a fair comparison with our previous work, however, our implementation of the path decomposition algorithm does not take advantage of the additional global paths. The results we present in Section 3.4 show the performance comparison up to the same path lengths for both implementations.

In this small example, the effect of local path computation is a reduction of the computational effort by a factor of 2. However, with an increasing path length, an increasing number of components, and an increasing number of synchronized transitions, the effect grows exponentially. In the following section, we introduce an algorithm for efficient computation of rewards for global paths from the results of local subpaths.

3.3.2 Algorithms for Efficient Composition of Subpaths

We have shown how paths can be decomposed into subpaths and how the subpath values are computed. In order to gain the benefits of the path-composition algorithm, we must have an effective strategy for exploring the subpaths and composing them efficiently. There are several strategies for exploring and computing the basic subpaths. Either they may all be precomputed before any computation of reward bounds begins, or they may be computed when required during the computation of the bounds. In both cases, the storage complexity is combinatorial in the number of subpaths. With so many paths to consider, the selection of the valid subpaths to compose is also expensive. We now describe an efficient algorithm for precomputing and composing the subpaths.

Instead of computing and storing all of the subpaths en masse, we can compute and store sets of them in stages. At each stage, we identify a set of subpaths that are *composable* with each other and compute only those. Subpaths are composable when they have the same longest common subsequence (LCS) [52] of synchronized transitions. In order to simplify the computation, we assume that the synchronized transitions cannot be transposed past each other, so that the LCS is unique. Thus, we can partition the set of all subpaths into classes of composable subpaths, with each class characterized by a sequence of synchronized transitions. For example, suppose $t_1, t_2, t_3 \in \mathcal{TS}$ are synchronized transitions. Then, the class of composable subpaths that have the sequence $\langle t_1, t_2, t_3 \rangle$ is $\{{l_j}^* \circ t_1 \circ {l_j}^* \circ t_2 \circ {l_j}^* \circ t_3 \circ {l_j}^*\}$, where ${l_j}^*$ denotes zero or more occurrences of the local transition of component j $(1 \leq j \leq J)$.

More formally, given a sequence of synchronized transitions $\langle t_1, \ldots, t_n \rangle$ and a maximum subpath length $L = n + K$, the class of composable subpaths corresponding to the sequence is

$$\begin{aligned} \mathcal{CS}(\langle t_1, \ldots, t_n \rangle, L) = & \bigcup_{j=1}^{J} \bigcup_{k_0^{(j)}+\ldots+k_n^{(j)}=0}^{K} \\ & \left\{ \boldsymbol{\pi}^{(j)} = {l_j}^{k_0^{(j)}} \circ t_1 \circ {l_j}^{k_1^{(j)}} \circ t_2 \circ {l_j}^{k_2^{(j)}} \circ \ldots \circ t_n \circ {l_j}^{k_n^{(j)}} \right\}. \end{aligned} \tag{3.14}$$

For each subpath $\boldsymbol{\pi}^{(j)} \in \mathcal{CS}(\langle t_1, \ldots, t_n \rangle, L)$, we need to precompute and store only one or two real values, namely $\nu^{(i)}(\boldsymbol{\pi^{(j)}})$ and $\xi^{(i)}(\boldsymbol{\pi^{(j)}})$. The probability of the subpath is given by

$$Prob^{(j)}(K) = \frac{\lambda_{l_j}^K}{\Lambda^K}, \tag{3.15}$$

and the probability of the synchronized transitions equals

$$ProbS(t_1 \ldots t_n) = \frac{\prod_{k=1}^{n} \lambda_{t_k}}{\Lambda^n}. \tag{3.16}$$

Each path that is composed from some subpaths in the class $\mathcal{CS}(\langle t_1, \ldots, t_n \rangle, L)$ can be considered as a canonical path that represents a set of equivalent paths. Computing the number of equivalent paths using the new algorithm is now straightforward (as compared to (3.10) in Section 3.2.2). The number of equivalent paths is simply

$$\prod_{i=0}^{|\langle t_1,\ldots,t_n \rangle|} \frac{\left(\sum_{j=1}^{J} k_i^{(j)}\right)!}{\prod_{j=1}^{J} \left(k_i^{(j)}!\right)}, \tag{3.17}$$

which follows from the number of possible reorderings of the local transitions between two immediate synchronized transitions.

Using (3.12), (3.15), (3.16), and (3.17), we can now compute the expected instantaneous reward for a model. Let Ψ be the set of all sequences of synchronized transitions. Then, the expected instantaneous reward is computed by

$$
\begin{aligned}
E[R_s] \;=\; \sum_{\psi\in\Psi}\sum_{m=|\psi|}^{\infty}\beta(\Lambda s,m) \sum_{\sum_{j=1}^{J}\sum_{i=0}^{|\psi|}k_i^{(j)}+|\psi|\,=\,m} \Bigg\{ \\
\prod_{i=0}^{|\psi|}\frac{\left(\sum_{j=1}^{J}k_i^{(j)}\right)!}{\prod_{j=1}^{J}\left(k_i^{(j)}!\right)}\cdot ProbS(\psi)\cdot\prod_{j=1}^{J}Prob^{(j)}\left(\sum_{i=0}^{|\psi|}k_i^{(j)}\right)\cdot \\
\prod_{j=1}^{J}\nu^{(j)}\left(l_j^{k_0^{(j)}},\psi_1,\ldots,\psi_{|\psi|},l_j^{k_{|\psi|}^{(j)}}\right)\Bigg\}
\end{aligned}
\tag{3.18}
$$

when the reward is computed from the product of component rewards. For an additive reward, we must use the summation of (3.12) instead of the last product in (3.18). The equations for the expected accumulated reward can be derived in a similar manner.

In summary, the algorithm works by generating the set of sequences of synchronized transitions. For each sequence, a class of composable subpaths is explored, and their values are precomputed using (3.12) and (3.15). Afterward, the subpaths are composed, and their values are used to compute the expected reward of a model. The subpaths in a class of composable subpaths may be explored using a depth-first strategy similar to that discussed in Section 3.2.2 to minimize the amount of memory used to store intermediate results. The algorithm is storage-efficient because during each stage of the computation, only two real values for each subpath are stored. Moreover, it is computationally efficient because each subpath, as a redundant computation across many paths, is computed only once and reused many times.

3.3.3 An Approach for Selecting Important Subpaths

Although there are many paths to consider, a large number of them often contribute little or no reward toward the computation of the bounds on the solution of a model. We can thus speed up the computation further by identifying important paths and discarding those that contribute little or nothing toward tightening of the bounds. While path selection has been considered by other researchers, our approach is new in that it bases the selection on additional information

available from the computed subpaths.

There are several subpath factors that directly affect the reward contribution of a path. With respect to (3.18), one of the main factors is the subpath reward $\nu^{(i)}(\boldsymbol{\pi}^{(i)})$. If a subpath has zero reward, all paths composed from it also have zero reward. Using this insight, we can improve the performance of the computation if we can identify efficiently those subpaths that contribute no reward and discard them from further computation.

Starting from (3.11) and (3.12), we note that $\nu^{(i)}(\boldsymbol{\pi}^{(i)})$ can be computed efficiently by first computing the projection of a component reward vector $\mathbf{r}^{(i)}$ onto component i. That yields a projected reward vector that can be cached and reused repeatedly. Next, when subpaths are being explored, we can compute their reward values efficiently by means of a scalar product of the subpath state distribution vector and the projected reward vector. In complexity terms, that incurs a cost of $\mathcal{O}(n_i)$ rather than $\mathcal{O}(n_i^2)$, where n_i is the size of the state space of component i.

We implement the approach by computing all projected reward vectors and caching them before any subpath is computed. As the subpaths are explored, the projected reward vectors are used to compute the subpath reward values. Those subpaths that contribute non-zero reward values are retained for composition with other subpaths; the rest are discarded immediately. When a zero-reward subpath is discarded, we keep the implementation simple by also discarding the successive subpaths that can be generated from the zero-reward subpath. Though the discarded successive subpaths may have non-zero reward values, their contributions toward tightening the bounds appear to be negligible in our experiments. In the next section, we show example results that are obtained using this approach.

3.4 Numerical Results

We evaluate our algorithm by studying its performance in analyzing two models with very different characteristics: a model of a distributed information service system adapted from the model in [48] and a model of a media multicast system inspired by the work of Chu et al [53]. In the former model, we evaluate the reliability and availability of the system; in the latter model, we evaluate the performability properties of the corresponding system. Moreover, the latter model has more components, a larger state space, and tighter coupling among the components. After describing the systems in detail below, we present the performance results and compare them with the results obtained using our earlier approach described in [1].

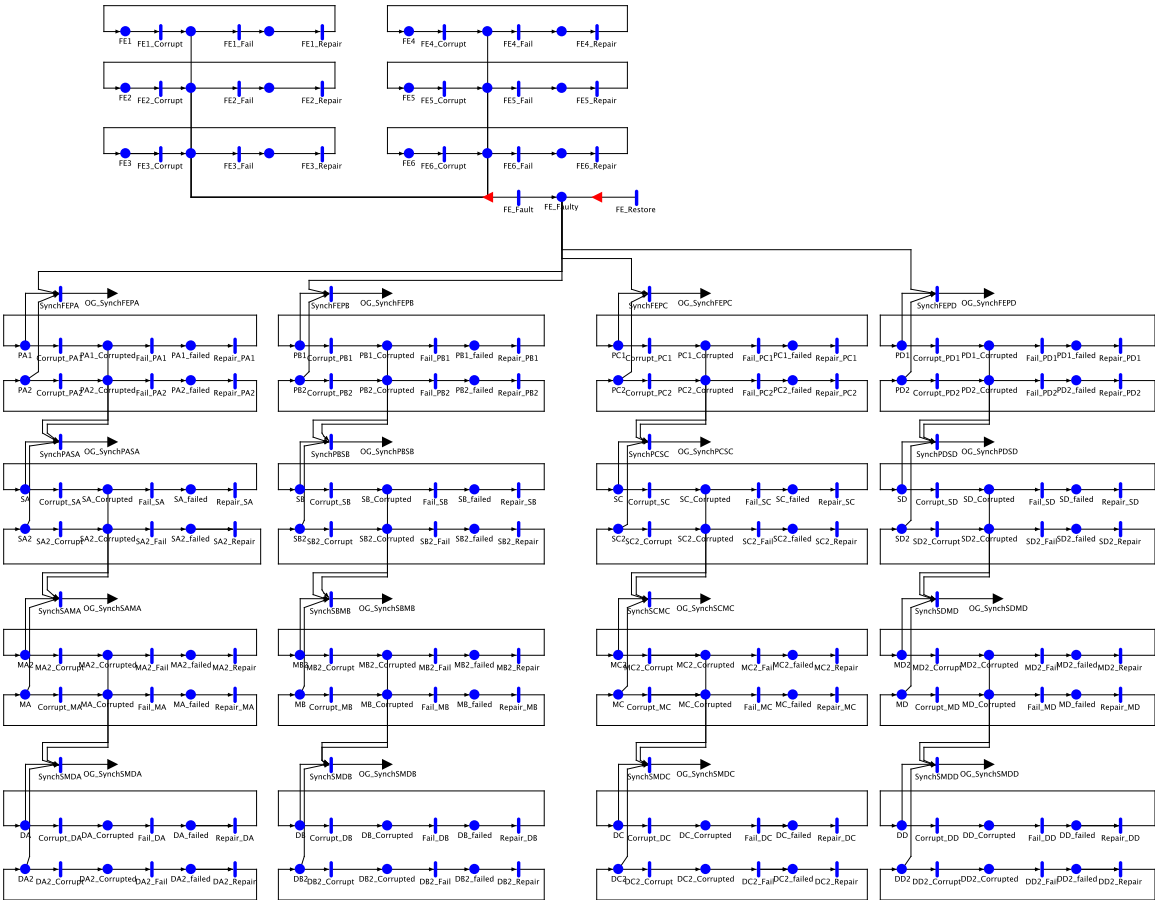


Figure 3.1: SAN model for computing the availability measure of a double-redundant distributed information service system.

3.4.1 Model Description of the Double-Redundant Distributed Information Service System

We augment the original model of the distributed information service system with synchronized transitions among the components to describe how faults are propagated through the system. In addition, we increase the number of front-end modules in order to model the occurrence of a fault only when a majority of the modules are corrupted. We also model double redundancy in the processing units by adding an additional module for every module in the original processing units. These additions quickly increase the size of our model, resulting in a model with approximately 2.7×10^{18} states, so large that it could not have been analyzed using traditional techniques, but can be using our approach.

The model consists of six front-end modules that interact with four processing units. Each processing unit consists of redundant components, including two processors, two switches, two

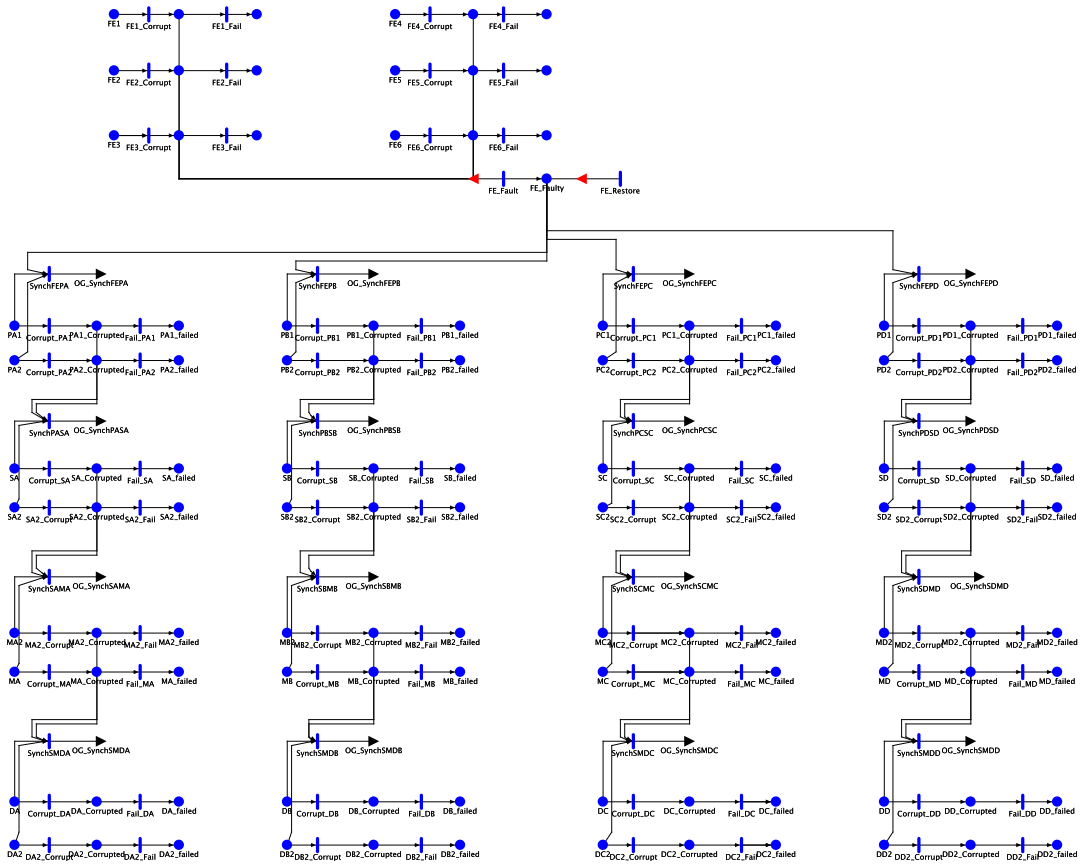


Figure 3.2: SAN model for computing the reliability measure of a double-redundant distributed information service system.

memory units, and two databases. Each of the components has its own repair facility. All of them go through the cycle of *Working*, *Corrupted*, *Failed*, and *Repaired*. The *stochastic activity network* (SAN) model of the system used for computing the availability measure is shown in Figure 3.1. The SAN model used for computing the reliability measure is shown in Figure 3.2. Notice that the latter model is different from the former in that it has no repair facilities in the individual modules, because it is a reliability model.

Fault propagation in the system is modeled as follows:

- When a majority of the six front-end modules are corrupted, the front-end is considered faulty, and it may propagate the error to any of the four processing units in which there are two working processors. Propagation occurs via the synchronized activities between the front-end and the processors in the processing units. The front-end or any of the processors

may disable the synchronized activities. After propagating the error to a processing unit, the front-end may remain in the faulty state and continue to propagate errors to other processing units until the majority fails or are repaired or there are no more processing units to which the error can be propagated.

- When both processors in a processing unit are corrupted, they both may propagate the error to their working switches via a synchronized activity. Any of the involved components may disable the synchronized activity. After the error propagation, the processors may remain in the corrupted state until they fail.

- When both switches of a processing unit are corrupted, they may propagate their errors to the working memory units via a synchronized activity. Any of these components may disable the activity. After propagating the error, the switches may remain in the corrupted state until they fail.

- When both memory units of a processing unit are corrupted, they may propagate their errors to the working databases via a synchronized activity. Any of these components may disable the activity. After propagating the error, the memory units may remain in the corrupted state until they fail.

We vary activity rates in the submodels and among the synchronized activities, so the resulting model does not have symmetries that would allow it to be lumped. Because of space constraints, we do not list the rates used for the model here. In total, the model has 5 submodels (modeling 38 components) and 4 synchronized activities. Because each component has three states, the state space of the whole model has $2 \times 3^{38} \approx 2.7 \times 10^{18}$ states. We computed the reliability of the system over the interval $[0, 1.0]$, the point availability at time 0.1, and the interval availability over the interval $[0.0, 0.1]$ when all components in the model were in the working state.

3.4.2 Model Description of the Media Multicast System

The SAN model of the media multicast system is shown in Figure 3.3. The model is parameterized by many variables, such that by varying the parameters for the activity rates and buffer sizes, we can measure the sensitivity of the system and the likelihood that it will experience buffer overflow. In addition, we can also compute the probability of having to flush the frame buffers when the system is corrupted. The model consists of a source (CMU) that multicasts

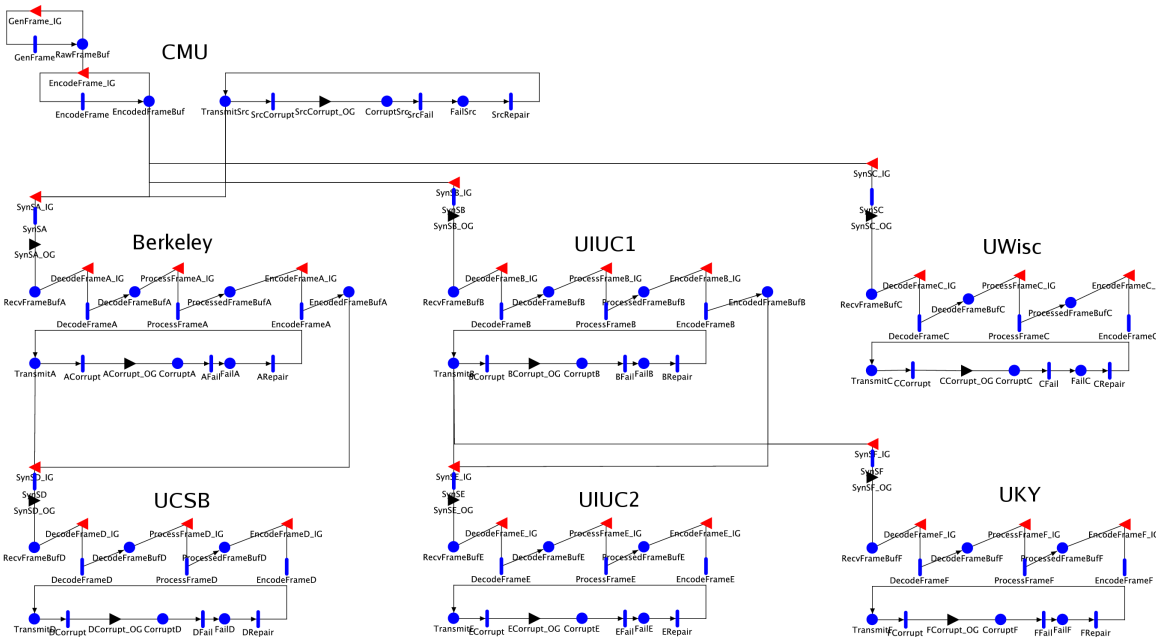


Figure 3.3: SAN model of a media multicast system.

frames to the clients Berkeley, UIUC1, and UWisc. Berkeley and UIUC1, in turn, multicast the frames further to UCSB, UIUC2, and UKY. Note that UWisc synchronizes only with CMU; Berkeley is more tightly connected with CMU and UCSB; and UIUC1 must synchronize with CMU, UIUC2, and UKY. The complete model has seven submodels and approximately 1.9×10^{20} states.

Frames are initially generated by the source, CMU. The tasks of the clients are to decode the frames, process them (perhaps adding more information to them), and encode them for further multicast. All of these components may be in any one of the operational, corrupted, or failed modes at any instant of time, and they all have their own repair facilities. They may transmit frames only when they are operational. When they are corrupted, their frame buffers are flushed, because the stored frames are presumably corrupted also. The transmitted frames are dropped when the clients' buffers are full.

Thus, the sensitivity of the system depends on the buffer sizes, the transmission rates, and the processing rates of the components. By varying these parameters, we can compute the probability of having a buffer overflow or buffer flushing at some time after the system has been in operation. In the next section, we present numerical results from our experimental evaluation. The results are not meant to be representative of any real system, since the parameters we used were not

Table 3.1: Numerical results for availability

Path Length	Lower Bound Point Availability	Upper Bound Point Availability	Lower Bound Interval Availability	Upper Bound Interval Availability	Basic Algorithm [1] Time (sec)	Path-composition Enhanced Algorithm Time (sec)
1	2.519650e-01	9.998975e-01	6.261971e-02	9.998443e-02	1.62	0.01
2	4.978311e-01	9.996229e-01	8.131310e-02	9.996355e-02	1.88	0.08
3	7.174581e-01	9.992548e-01	9.180509e-02	9.994597e-02	4.88	0.87
4	8.645992e-01	9.989260e-01	9.680362e-02	9.993480e-02	42.25	8.88
5	9.434621e-01	9.987056e-01	9.885818e-02	9.992906e-02	504.81	92.18
6	9.786854e-01	9.985875e-01	9.959795e-02	9.992657e-02	6135.23	934.25
7	9.921700e-01	9.985347e-01	9.983439e-02	9.992565e-02	—	9673.85

Table 3.2: Numerical results for reliability

Path Length	Lower Bound Reliability	Upper Bound Reliability	Basic Algorithm [1] Time (sec)	Path-composition Enhanced Algorithm Time (sec)
1	6.522987e-01	9.955666e-01	1.26	0.00
2	8.661349e-01	9.902090e-01	1.52	0.08
3	9.517685e-01	9.869717e-01	4.41	0.72
4	9.774886e-01	9.856676e-01	40.38	7.30
5	9.836687e-01	9.852736e-01	487.09	74.02
6	9.849062e-01	9.851784e-01	5990.44	759.56
7	9.851185e-01	9.851592e-01	—	7763.17

taken from a real system. They do show, however, that the parameters are interdependent and that our algorithm works correctly in computing the results for the varied parameters.

3.4.3 Experimental Evaluation

We conducted all of our experiments on a workstation that had the AMD Athlon XP 2700+ processor running at 2.17 GHz with 1.0 GB of RAM. The operating system was Red Hat Linux 9.0 with mounted file systems. We compiled our implementation using the compiler g++ 3.3 with optimization flag -O3 only.

For the model of the distributed information service system, the Möbius simulation results are $0.99883 \pm 2.11883414 \times 10^{-4}$ for the point availability, $0.099934 \pm 1.340263 \times 10^{-5}$ for the interval availability, and $0.98616 \pm 7.241023 \times 10^{-4}$ for reliability. These results were obtained at a 95% confidence level.

For the model of the media multicast system, the Möbius simulation results are $0.925 \pm 1.63 \times 10^{-2}$ for the probability that the system will experience a buffer overflow and $0.365 \pm 2.99 \times 10^{-2}$ for the probability that the system will have to flush its frame buffers due to a system failure. These results were also obtained at a 95% confidence level.

Table 3.3: Numerical results for the probability of buffer overflow

Path Length	Basic Algorithm Lower Bound	Basic Algorithm Upper Bound	Enhanced Algorithm Lower Bound	Enhanced Algorithm Upper Bound	Basic Algorithm [1] Time (sec)	Path-composition Enhanced Algorithm Time (sec)
1	1.492972e-01	9.962873e-01	1.492972e-01	9.962873e-01	0.21	0.00
2	3.341998e-01	9.840578e-01	3.341998e-01	9.840578e-01	0.25	0.02
3	5.345312e-01	9.644779e-01	5.339680e-01	9.639147e-01	0.89	0.19
4	6.978060e-01	9.437606e-01	6.958400e-01	9.417942e-01	10.42	2.05
5	8.045237e-01	9.273265e-01	8.007710e-01	9.235738e-01	152.80	22.71
6	8.627557e-01	9.168673e-01	8.574547e-01	9.115663e-01	2266.00	257.59

Table 3.4: Numerical results for the probability of buffer flushing

Path Length	Basic Algorithm Lower Bound	Basic Algorithm Upper Bound	Enhanced Algorithm Lower Bound	Enhanced Algorithm Upper Bound	Basic Algorithm [1] Time (sec)	Path-composition Enhanced Algorithm Time (sec)
1	1.177836e-01	9.647736e-01	1.177836e-01	9.647736e-01	0.15	0.00
2	2.146505e-01	8.645084e-01	2.148119e-01	8.646698e-01	0.18	0.02
3	2.903990e-01	7.203457e-01	2.909466e-01	7.208933e-01	0.62	0.17
4	3.348248e-01	5.807794e-01	3.358348e-01	5.817894e-01	7.09	1.89
5	3.556691e-01	4.784719e-01	3.570485e-01	4.798513e-01	103.90	20.74
6	3.638190e-01	4.179306e-01	3.654202e-01	4.195318e-01	1564.00	229.20

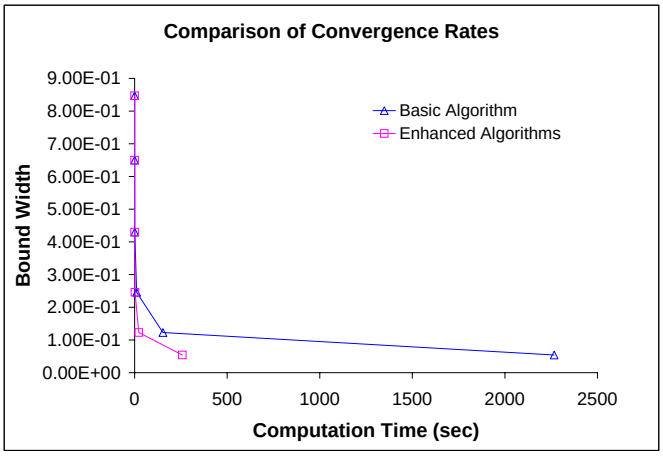
Tables 3.1 and 3.2 show the results for the availability and reliability measures, respectively, of the model of the distributed information service system calculated using our path-based approach. Note that the lower and upper bounds for each measure converge as the path length increases, because more paths are computed. Although the path-selection approach discards zero-reward subpaths and their successive subpaths, the bound values for this particular model are not affected up to the seventh significant digits in any of the experiments we performed. Column *Basic Algorithm Time (sec)* lists the time taken to evaluate the model through the use of the path-based approach described in [1]. Column *Path Decomposition Enhanced Algorithm Time (sec)* lists the time taken using of our new algorithm, which makes use of the path-decomposition and path-selection schemes described in this chapter. As shown in the time columns for both the availability and reliability results, our new algorithm achieves approximately 80% performance improvement relative to the previous algorithm. As the path length gets longer, the algorithm performs better. For example, at the path length of 6 for the availability results, it achieves almost 85% improvement; at the same path length for the reliability results, it achieves 87% improvement. We do not have the timing result for the basic approach at the path length of 7, because it takes too long to compute.

Tables 3.3 and 3.4 show the results for the probabilities of buffer overflow and buffer flushing,

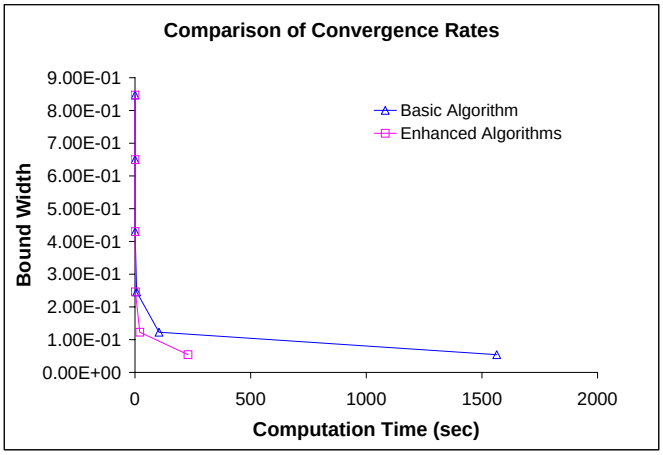
respectively, for the model of the media multicast system. For this model, the values of the bounds computed by the basic approach and by the new algorithm differ somewhat. We list the values of both bounds in the tables for comparison. In order to understand better the rates of convergence of the bounds, we provide a graphical comparison of the convergence rates in Figure 3.4.

3.5 Summary

Even though existing path-based techniques have been shown to be effective in reducing the amount of memory necessary to analyze very large models, they are still limited in the size of problems they can solve, due to the large number of paths that often need to be explored to obtain tight bounds on a measure. This chapter presented a novel approach for computing paths based on the idea of path composition. Instead of computing paths directly, the approach first computes possible subpaths for each component of a model. The sets of subpaths are then composed to explore many paths simultaneously. Effectively, this approach eliminates redundant computation expended in computing common subpaths found across multiple paths. Furthermore, we showed how a path-selection approach works seamlessly with the path-composition algorithm to find important subpaths efficiently. As a result, we were able to achieve a speedup of 6.6 to 8.8 times for two benchmark models. To the best of our knowledge, our work is the first to propose the use of path composition for the analysis of Markov models. These improvements make it feasible to evaluate efficiently models of practical systems that are significantly larger than could previously be handled.



(a) Buffer Overflow



(b) Buffer Flushing

Figure 3.4: Comparison of convergence rates between the basic algorithm and the enhanced algorithm.

CHAPTER 4

THE PATH-SELECTION APPROACH

4.1 Introduction

In this chapter, we present three path-selection approaches for identification and evaluation of important paths. The ability to identify important paths is instrumental in solving a model quicker and it makes solving even larger models feasible. The algorithms for the approaches are discussed, and their implementation is presented through pseudocode. Then their benefits are evaluated with respect to their usefulness in computing solutions for models of availability, reliability, and performability. Finally, the strengths and limitations of the approaches are discussed in the summary of the chapter.

4.1.1 Motivation and Problem Statement

As the previous chapters have shown, the structured path-based approach is useful for solving very large Markov models. The models are solved through the evaluation of many paths over which the models may evolve. In a sense, the approach resembles a systematic simulation of the evolutionary trajectories of actual systems to measure their performance characteristics, such as availability, reliability, and performability. Evaluation of paths helps overcome the large memory complexity inherent in large models, which many other analytical techniques simply cannot manage.

The benefit of the structured path-based approach extends beyond its ability to solve very large models. Usually, in system specification, a performance characteristic is defined in terms of a range of acceptable values. For example, it may be specified that a system must be available *at least* 99.999% of the time instead of being available *exactly* 99.999% of the time. The path-based approach solves a model and presents its solution as lower and upper bounds around the exact solution. The bounds can be used directly to determine whether the specified range is satisfiable.

To make the approach even more useful and practicable, we need to improve its performance dramatically so as to be able to compute even tighter bounds. Computing tight bounds generally requires evaluation of many paths. The number of paths increases exponentially with the lengths of the paths. For example, the model of a distributed information service system described in Section 4.3 has 14 different transitions. In order to analyze it for paths that have lengths up to 10, the structured path-based techniques (described in the previous section) has to evaluate 14^{10} or approximately 290 billion paths. Since the evaluation of a path of length l requires at least l vector-matrix multiplications, in which the dimension of the matrix is on the order of $(n_i \times n_i)$, evaluation of all these paths would clearly take an unacceptably long time. The purpose of this chapter is to present three path-selection techniques for identification of important paths. The approaches are very effective for computing tighter bounds in a shorter amount of time, because they selectively evaluate a much smaller number of only the most important paths.

4.1.2 Algorithmic Review of the Structured Path-based Approaches

We review how paths are explored in the structured path-based approach described in [1]. The approach does not employ any important-path selection technique. Paths are explored and evaluated in depth-first order. Memory complexity is very low, since it is on the order of the length of a path. The approach has poor performance, however, because it potentially has to evaluate many unimportant paths.

```
StructuredPathApproach(π)
 1  π' := π_0
 2  while (|π'| < max_length)
 3      UpdateStateVectors(π')
 4      UpdateRewardBounds(π')
 5      foreach (t ∈ Transitions)
 6          π' := π' ∘ t
 7          StructuredPathApproach(π')
```

Figure 4.1: Algorithm for exploring and evaluating paths in the structured path-based approach.

The algorithm for path exploration in the approach is shown in Figure 4.1. Paths are exhaustively explored up to a user-specified maximum length `max_length` (Line 2). As paths are explored, they are used to compute bound values in Lines 3-4. New paths are computed from existing paths through appending of various transitions (Lines 5-7).

4.2 Technical Description of the Path-Selection Approaches

In this section, we present three approaches for identifying important paths and selectively choosing them for evaluation. The goal is to evaluate a much smaller, but more relevant, set of paths so as to tighten the bounds quickly. The first two approaches select paths based on their reward values and their probabilities, respectively. Although these approaches are effective in helping to improve performance, they do have their limitations. The third approach integrates the first two approaches to overcome their individual limitations. All of the approaches manage their memory usage to avoid thrashing in the memory system. We begin by describing the memory management technique. Afterward, we present the path-selection approaches and discuss their limitations.

4.2.1 Memory Management in the Path-Selection Approaches

The path-selection approaches benefit when there are many paths from which they can choose those to evaluate. On the other hand, the number of paths that they explore can grow exponentially, so that they use up all available storage space. In order to avoid thrashing in the memory system, we use a memory management technique whereby a user specifies the lower and upper memory usage thresholds to confine the approaches so that they use only a prescribed amount of storage. The upper threshold specifies the maximum amount of storage that the approaches may use. When the upper threshold is exceeded, the memory manager is initiated to discard the least important paths and to recover the storage space used to store them. Once initiated, the memory manager continues to reclaim memory until the lower threshold is reached. Then control is returned to the path-selection approaches so that they can proceed to evaluate the remaining paths and explore further other paths until the maximum threshold is exceeded again.

We use a priority queue to prioritize the explored paths from most important to least important. When the memory manager is initiated, the paths in the queue are sorted from least important to most important. We use the heap sort to achieve optimal runtime, and it is on the order of $\mathcal{O}(n \log n)$, where n is the number of paths in the queue.

When a path is discarded, memory is reclaimed from the data structures that are used to represent the path. In addition, when a set of paths that share some common data structures (e.g., the component state distribution vectors) are discarded, the memory for the shared data structures is also reclaimed. Note further that when a path is discarded, paths that are derived

successively from the discarded path are not explored.

4.2.2 Path Selection Based on Reward Values of Paths

The path reward values contribute directly toward the computation of the upper and lower bounds on the solution of a model. They are also the most expensive to compute, since computing them requires vector-matrix multiplication operations. Thus, when we select paths based on their reward values, we attempt to minimize as much as possible the computation of the vector-matrix multiplications. We now describe a technique for computing the reward value of a path inexpensively. Then we present a path-selection approach based on the technique for selecting paths according to their reward values.

Let $\boldsymbol{\pi}$ be a path. The reward after the transition of $\boldsymbol{\pi}$ is computed by

$$R(\boldsymbol{\pi}) = \bigotimes_{j=1}^{J} \mathbf{p}_0^{(j)} \left(\prod_{k=1}^{|\boldsymbol{\pi}|} \mathbf{P}_{\boldsymbol{\pi}(k)} \right) \bigodot_{j=1}^{J} \mathbf{r}^{(j)} \ , \tag{4.1}$$

where the operator $\odot$ may be either $\otimes$ or $\oplus$ depending on whether the reward vector is computed by the Kronecker product or the Kronecker sum, respectively. Using (4.1), we can compute reward for the extended path $\boldsymbol{\pi} \circ a$ (i.e. extension by transition a) by

$$\begin{aligned} R(\boldsymbol{\pi} \circ a) &= \bigotimes_{j=1}^{J} \mathbf{p}_0^{(j)} \left(\prod_{k=1}^{|\boldsymbol{\pi}|} \mathbf{P}_{\boldsymbol{\pi}(k)} \right) \cdot \mathbf{P}_a \bigodot_{j=1}^{J} \mathbf{r}^{(j)} \\ &= \bigotimes_{j=1}^{J} \mathbf{p}^{(j)}[\boldsymbol{\pi}^{(j)}] \cdot \mathbf{P}_a \bigodot_{j=1}^{J} \mathbf{r}^{(j)} \end{aligned} \tag{4.2}$$

Note that (4.2) is essentially a projection of the component state-distribution vectors onto the transformed component reward vectors.

Equation (4.2) provides the basis for constructing an approach for selecting paths based on their reward values. Since the component reward vectors do not change during the computation of the bounds, we can precompute their transformed counterparts for all transitions in a model. The precomputation is inexpensive because it is done only once for each transition and it involves only a few vector-matrix operations. After the transformed reward vectors have been precomputed, we can compute the reward for an extended path by computing the scalar projection of the component state-distribution vectors onto the transformed reward vectors. Note

```
PrecomputeTransformedRewardVectors()
 1  foreach t ∈ Transitions
 2      if t ∈ {l_1, ..., l_J}
 3          r̃^(j)_{P_l^(j)} := P_l^(j) · r^(j)
 4      else if t ∈ T_S
 5          r̃^(j)_{E_t^(j)} := E_t^(j) · r^(j)
 6      else if t = t̄_j ∧ t ∈ T_S
 7          foreach i < j
 8              r̃^(i)_{D_t^(i)} := D_t^(i) · r^(i)
 9          r̃^(j)_{Ē_t^(j)} := Ē_t^(j) · r^(j)
10      else
11          r̃^(j)_{I_{n_j}} := I_{n_j} · r^(j)
```

Figure 4.2: Algorithm for computing transformed reward vectors.

```
ExplorePath(π, Q^P)
1  foreach t ∈ Transitions
2      π' := π ∘ t
3      π'.reward := ComputePathReward(π')
4      π'.nmEquiv := ComputeNumEquivPaths(π')
5      π'.weight := π'.nmEquiv * π'.reward
6      Push(π', Q^P)
```

Figure 4.3: Algorithm for path exploration based on path reward values.

that computing the reward value of an extended path using the projection method incurs a cost of $\mathcal{O}(J \cdot n_i)$ as compared to $\mathcal{O}(J \cdot n_i^2)$, a huge saving when many paths must be evaluated.

The pseudocode for computing the transformed reward vectors and caching them is shown in Figure 4.2. For a local transition, only the reward vector of the component affected by the transition needs to be transformed (Lines 2-3). That requires only one matrix-vector multiplication. For a synchronized transition, the reward vectors of all components affected by the transition need to be transformed (Lines 4-5). In this situation, at most J matrix-vector multiplications are required. For a disabled synchronized transition in which component j disabled the transition, all reward vectors $\mathbf{r}^{(i)}$, $i < j$, are transformed by the matrices $\mathbf{D}_t^{(i)}$, and the reward vector $\mathbf{r}^{(j)}$ is transformed by matrix $\overline{\mathbf{E}}_t^{(j)}$ (Lines 6-9).

The pseudocode for the path-exploration algorithm is shown in Figure 4.3. Given a path $\boldsymbol{\pi}$ and a priority queue $\mathcal{Q}^P$, the algorithm computes a new path $\boldsymbol{\pi}'$ by appending a transition t to $\boldsymbol{\pi}$.

```
SelectPathsByReward()
 1  PrecomputeTransformedRewardVectors()
 2  Q^P := {π_0}
 3  while ((Q^P ≠ ∅) ∧
        (Ub(E[R_s]) − Lb(E[R_s])) ≥ user_prec)
 4     π := Pop(Q^P)
 5     UpdateStateVectors(π)
 6     UpdateRewardBounds(π)
 7     ExplorePath(π, Q^P)
 8     if (MemUsage ≥ MEM_ULIMIT)
 9        ReclaimMemory()
```

Figure 4.4: Algorithm for selecting paths based on their reward values.

Then the reward value of $\boldsymbol{\pi}'$ and the number of paths that are equivalent to it are computed (Lines 3-4) in order to determine its priority among other explored paths. Since the individual paths have different numbers of equivalent paths, it is necessary to compute the numbers of equivalent paths to determine which paths are more important than others. Then, the weight of $\boldsymbol{\pi}'$ is computed by the product of those values. Finally, $\boldsymbol{\pi}'$ can be enqueued and prioritized according to its weight.

Figure 4.4 presents the algorithm for selecting paths according to their reward values. Paths are explored and evaluated in a tree-traversal manner. The algorithm is based on the use of a priority queue. The queue is used for prioritizing paths as they are being explored but not yet evaluated. Recall that the full evaluation of a path incurs a cost on the order of $\mathcal{O}(J \cdot n_i^2)$, whereas estimating the importance of the path based on its reward value using the projection method during the path exploration stage incurs only the cost of $\mathcal{O}(J \cdot n_i)$. We take advantage of this optimization in the algorithm as follows. First, the transformed reward vectors are precomputed, and the priority queue is initialized to contain an initial path (Lines 1-2). Next, while the queue is not empty and the bound width does not meet a user's specification, new paths are repeatedly explored and evaluated (Lines 3-9). At Line 4, the most important path among those that have been explored is removed from the queue for evaluation. The path is fully evaluated at Line 5 and incurs the cost of $\mathcal{O}(J \cdot n_i^2)$. After it has been evaluated, all successive paths derived from it are explored and prioritized (Line 7). When the maximum memory usage threshold is exceeded, memory is reclaimed through deletion of the least important paths in the queue (Lines 8-9). It should be emphasized that those paths that are deleted due to memory reclamation incur only the smaller cost of $\mathcal{O}(J \cdot n_i)$.

The path-selection approach described above has the following shortcomings for a certain class of models. For some availability models (in which the components have their own repair facilities), many long paths yield the same reward values as the short ones. It is important to note, however, that beyond some length the probabilities of the longer paths are generally smaller than those of the shorter ones, even when their reward values are equal. Nevertheless, they are enqueued at the top of the queue as if they had high priority, because the priority queue is used and it does not distinguish between two paths that have the same reward values. Thus, some paths may be falsely classified as having higher priority than other paths that truly contribute more toward tightening the bounds. Consequently, convergence of the bounds for these models may be slow. Furthermore, when there is not enough available storage, some truly important paths may be deleted when memory is reclaimed while the falsely prioritized paths are kept in the queue. That can lead to computation of bounds that are not as tight as they could have been. We present results demonstrating those situations in Section 4.4.

4.2.3 Path Selection Based on Probability Values of Paths

The probability of a path is characterized by two factors. The first factor is the probability distribution of the number of events occurring in the time interval $[0, s)$, where the number of events corresponds to the number of transitions in the path and is its length. That factor is computed by the Poisson density $\beta(\Lambda s, k)$, where $\Lambda = \sum_{i=1}^{J} \lambda_{l_i} + \sum_{t \in \mathcal{T}_S} \lambda_t$ and k is the length of the path. The second factor is the product of the probabilities of the occurrences of the individual events. The probability of the occurrence of an event x is computed by λ_x/Λ, where x corresponds to a transition in the model and λ_x is the rate of the transition. The product of these two factors yields the probability of a path. Given the path $\boldsymbol{\pi}$, its probability is computed as follows:

$$P(\boldsymbol{\pi}) = \beta(\Lambda s, |\boldsymbol{\pi}|) \cdot \prod_{k=1}^{|\boldsymbol{\pi}|} \frac{\lambda_{\boldsymbol{\pi}(k)}}{\Lambda} \tag{4.3}$$

The algorithm for selecting paths based on their probabilities is similar to the one for selecting paths based on their reward values. The main difference is in how the weight of a path is computed in the path exploration algorithm. The pseudocode for the algorithm is shown in Figure 4.5. For a similar reason that the numbers of equivalent paths are used to compute the weights of paths in the reward-based approach, the numbers of equivalent paths must also be

```
ExplorePath(π, Q^P)
 1  foreach t ∈ Transitions
 2      π' := π ∘ t
 3      π'.probability := ComputePathProbability(π')
 4      π'.nmEquiv := ComputeNumEquivPaths(π')
 5      π'.weight := π'.nmEquiv * π'.probability
 6      Push(π', Q^P)
```

Figure 4.5: Algorithm for path exploration based on path probability values.

used in this algorithm.

The probability-based path-selection approach is not effective for analyzing rare-event models. Those models typically have a few low-probability paths that have high reward values. Since the less probable paths could be deleted when memory is reclaimed, the computed bounds would not be tight.

4.2.4 The Integrated Path-selection Approach

Path reward and probability values affect the lower and upper bounds differently. From Equations (2.14) and (2.15), we observe that the reward value of a path has a more direct effect in raising the lower bound. The upper bound is affected largely by the probability value of the path. When a solution is computed for a model, at some phases of the analysis, one set of paths may yield large reward values, while at other phases, another set of paths have large probability values. We now present an integrated approach that adapts to the phases by switching dynamically during analysis between the reward-based and probability-based path-selection approaches. The goals are to overcome some of the shortcomings of the two individual approaches and to compute tight bounds.

Figure 4.6, for example, shows how probability values vary with the length of a path. The Poisson density curve shows the distribution of the numbers of events in the paths. The event sequence probability curve shows the probabilities of particular sequences of transitions that make up the paths. As shown in the figure, both curves have peak values, and they taper off for long path lengths. Note that for each model, there is one Poisson density curve, but there are many event sequence probability curves for the many individual paths. Thus, to be effective in tightening the bounds, the integrated approach must use the probability-based path-selection approach during the phases when high-probability paths are explored. Similarly, when it has

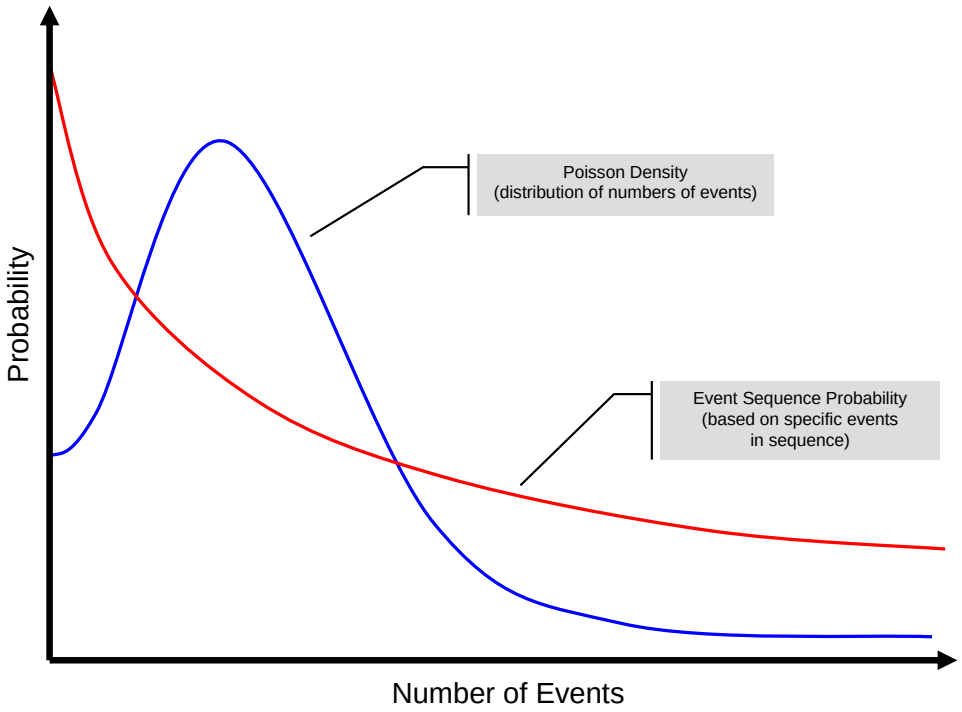


Figure 4.6: Path probability curves.

determined that a set of high-reward paths have been explored, and that they help to tighten the bounds quickly, the integrated approach should know to switch to path selection based on reward values.

In dynamically determining whether to use the probability-based or reward-based path-selection approach, it is important to consider the rates of convergence of the two bounds. The rates of convergence can be estimated from the slopes of the bounds. During analysis, the bound that has the steeper slope indicates that using the corresponding path-selection approach yields faster convergence toward the exact solution. Figure 4.7 illustrates the concept. Thus, if it is determined that the upper bound has the steeper slope, then the paths that are in the priority queue and those that will be explored should be prioritized and selected according to their probability values. Similarly, if it is determined that the lower bound has the steeper slope, then the paths should be prioritized and selected according to their reward values.

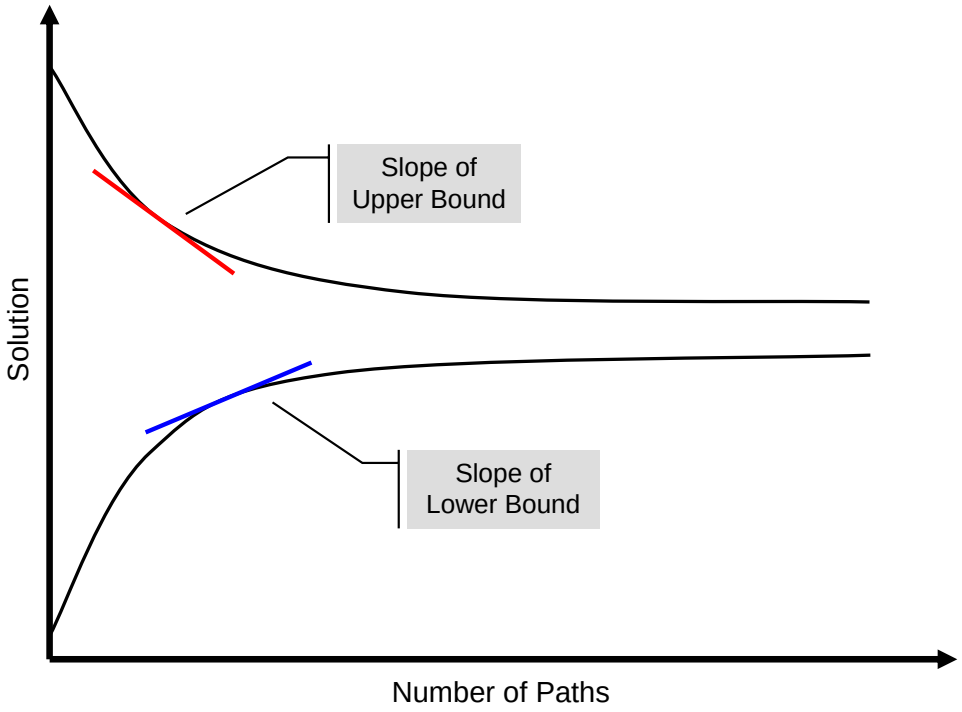


Figure 4.7: Path probability curves.

```
ComputeLowerBoundSlope(n)
 1  InitialReward = ComputeLowerBoundReward()
 2  PrioritizePathsByRewardValues()
 3  for i := 1 to n
 4      π := Pop(Q^P)
 5      UpdateStateVectors(π)
 6      UpdateRewardBounds(π)
 7      ExplorePath(π, Q^P)
 8      if (MemUsage ≥ MEM_ULIMIT)
 9          ReclaimMemory()
10  LowerBoundSlope := (ComputeLowerBoundReward() - InitialReward) / n
11  return (LowerBoundSlope)
```

(a)

```
ComputeUpperBoundSlope(n)
 1  InitialReward = ComputeUpperBoundReward()
 2  PrioritizePathsByProbabilityValues()
 3  for i := 1 to n
 4      π := Pop(Q^P)
 5      UpdateStateVectors(π)
 6      UpdateRewardBounds(π)
 7      ExplorePath(π, Q^P)
 8      if (MemUsage ≥ MEM_ULIMIT)
 9          ReclaimMemory()
10  UpperBoundSlope := (InitialReward - ComputeUpperBoundReward()) / n
11  return (UpperBoundSlope)
```

(b)

```
IntegratedPathSelectionApproach()
 1  PrecomputeTransformedRewardVectors()
 2  Q^P := {π_0}
 3  LowerBoundSlope := ComputeLowerBoundSlope(n)
 4  UpperBoundSlope := ComputeUpperBoundSlope(n)
 5  while ((Ub(E[R_s]) − Lb(E[R_s])) ≥ user_prec)
 6      if (LowerBoundSlope > UpperBoundSlope)
 7          LowerBoundSlope := ComputeLowerBoundSlope(n)
 8      else
 9          UpperBoundSlope := ComputeUpperBoundSlope(n)
```

(c)

Figure 4.8: Integrated algorithm for selecting paths based on their reward and probability values.

Figure 4.8 presents the algorithm for the integrated path-selection approach in pseudocode. The main algorithm is shown in 4.8(c). Initially, n paths are prioritized according to their reward values and evaluated to estimate the slope of the lower bound (Line 3). Then, another n paths are prioritized according to their probability values and are evaluated to estimate the slope of the upper bound (Line 4). Note that the order of initialization may affect the convergence of the bounds and, consequently, the bound width. Thereafter, the paths are prioritized according to reward or probability values based on whether the lower or the upper bound has the steeper slope, respectively. For each interval between two estimations of the slopes, n paths are evaluated. For more dynamic adjustment of the approach to changes in the convergence rates of the bounds, n may be set to smaller values. Setting n to smaller values, however, incurs larger performance cost due to more frequent sorting and prioritizing of the paths.

Figures 4.8(a) and 4.8(b) show the algorithms for computing the slopes of the lower and upper bounds, respectively. They are identical except for Lines 1, 2, 10, and 11, which are specific to each algorithm. For the algorithm to compute the slope of the lower bound, for example, Line 1 computes the initial value of the lower bound at the beginning of an n-path interval. Line 2 sets up the algorithm to prioritize paths according to their reward values. Lines 3-9 select and evaluate the n most important paths and, in the process, update the bounds. Finally, Lines 10-11 compute and return the slope.

4.2.5 Baseline Approach for Comparison with the Path-Selection Approaches

In order to evaluate the effectiveness of the path-selection approaches, we compare their performance to that of the following baseline approach. We equip the baseline approach with the memory management module (as we do with the path-selection approaches) so that it does not thrash the memory system. The baseline approach, however, does not evaluate paths selectively. Instead, it evaluates paths in a FIFO order. That is, a FIFO queue is used in place of the priority queue.

Figure 4.9 shows the algorithm for the baseline approach. $\mathcal{Q}$ is a FIFO queue. In Figure 4.9(a), at Line 3, new paths are explored and enqueued at the back of the queue. In Figure 4.9(b), the paths are evaluated from the front of the queue.

In the next section, we present the models that we used to evaluate these path-selection approaches. Afterward, we present quantitative results of the evaluation along with a discussion

```
ExplorePath(π, Q)
1  foreach t ∈ Transitions
2      π' := π ∘ t
3      Push_Back(π', Q)
```
(a)

```
BaselineApproach()
1  Q := {π_0}
2  while ((Q ≠ ∅) ∧
       (Ub(E[R_s]) − Lb(E[R_s])) ≥ user_prec)
3      π := Pop_Front(Q)
4      UpdateStateVectors(π)
5      UpdateRewardBounds(π)
6      ExplorePath(π, Q)
7      if (MemUsage ≥ MEM_ULIMIT)
8          ReclaimMemory()
```
(b)

Figure 4.9: Algorithm for baseline approach for evaluating paths.

of the insight we gained. Finally, we summarize our work on path-selection approaches in Section 4.5.

4.3 Description of Experimental Models

We evaluate the path-selection approaches in analyzing three models that span a range of different characteristics. The approaches are used to compute metrics of availability, reliability, and performability.

4.3.1 Availability and Reliability Models

Figures 2.1 and 2.2 show the availability and reliability models of a distributed information service system, respectively, that we used to evaluate the performance of the path-selection approaches. More detailed descriptions of the models can be found in Section 2.7.1. Each model consists of a front-end unit and four processing units. Faults in the front end can propagate to the processing units. That is modeled using shared activities between the front end and the processing units.

In the availability model, the components in the front end and the processing units have their own repair facilities, as depicted by the activities at the end of the components and the loops back to the places at the front of the components. The reliability model does not have repair facilities as in the availability model. The experimental results for the availability model are shown in Section 4.4.1, and the results for the reliability model are shown in Section 4.4.2.

4.3.2 Performability Model

The performability model is shown in Figure 3.3. The model is used to study the joint characteristics of performance and reliability of a multimedia multicast system. The system consists of a source that multicasts media frames to three clients that, in turn, process and multicast the frames on to their clients. Depending on the transmission and processing rates and the buffer sizes, buffer overflow may occur in the clients. Furthermore, the source and clients are prone to failures. When failures occur, the affected buffers must be flushed. Note that the source and clients have their own repair facilities. Thus, after experiencing failures, they can repair themselves to get back to the working state. The model is used to measure the probabilities of having buffer overflow and of having to flush buffers.

4.4 Experimental Results and Discussion

We now present the result of the evaluation of the three path-selection approaches in solving the availability, reliability, and performability models, respectively. Because there is a large amount of information to present and summary graphs would omit important details, we have chosen to show the results for each approach individually to help give deeper insight into the behaviors of the approaches. In addition, we present combined graphs for comparing the relative performance of the approaches.

We use three performance metrics for evaluating the approaches:

- **Number of Evaluated Paths**: This metric refers to those paths that were actually evaluated by the approach rather than those that were only explored. The former incurs the full cost of $\mathcal{O}(J \cdot n_i^2)$ to perform the vector-matrix multiplication operations.
- **Elapsed Time**: This metric is the sum of the user and system times (in seconds) as reported by the *Unix* `getrusage` system call.
- **Bound Width**: This metric is computed by the difference between the upper and lower bounds on the transient measures of interest.

For each model and each path-selection approach, we present three graphs comparing

(a) Number of Evaluated Paths vs. Bound Width,

(b) Elapsed Time vs. Bound Width, and

(c) Number of Evaluated Paths vs. Elapsed Time.

After the graphs for the individual approaches have been presented, we present the combined graphs to compare the three approaches directly. We evaluated the availability and reliability models at two transient time points, 10.0 and 100.0, to determine whether the approaches are sensitive to different time points.

We use the following general evaluation criteria in interpreting the results in the three types of graphs mentioned above, respectively:

(a) The smaller the number of paths that are evaluated to reach a certain bound width, the better the approach. Similarly, the better approach can compute smaller bound widths by evaluating a fixed number of paths.

(b) Give a certain bound width, the better approach takes less time to compute it. Similarly, given a fixed amount of computation time, the better approach can compute a smaller bound width.

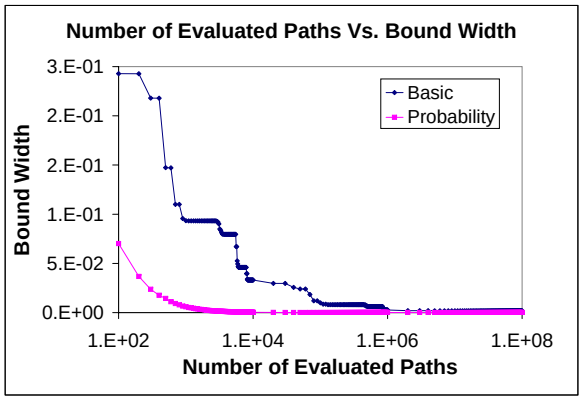
(c) The better approach can compute more paths in less time.

All of the experiments were conducted on a workstation that had the AMD Athlon XP 3700+ processor running at 2.2 GHz with 2.0 GB or RAM. The operating system was Fedora Core 4 with mounted file systems. We compiled our implementation using the compiler g++ 3.4.4 with optimization flag -O3 only.

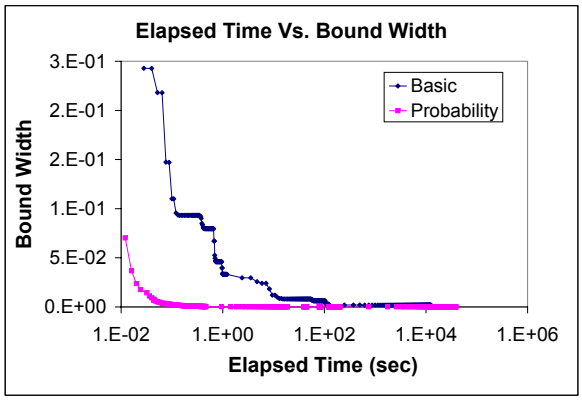
4.4.1 Availability Model

The results for the evaluation of the availability model at transient time point 10.0 are shown in Figures 4.10 – 4.13. As shown in Figure 4.10, the probability-based selection approach is able to compute tight bounds quickly. The graph of the bound width for the approach essentially flattens after approximately 2.0×10^3 paths have been computed, which took less than 0.1 second to compute. The graph for the bound width of the reward-based approach, however, does not flatten out until about 2.0×10^6 paths have been computed, which took more than 220 seconds to compute.

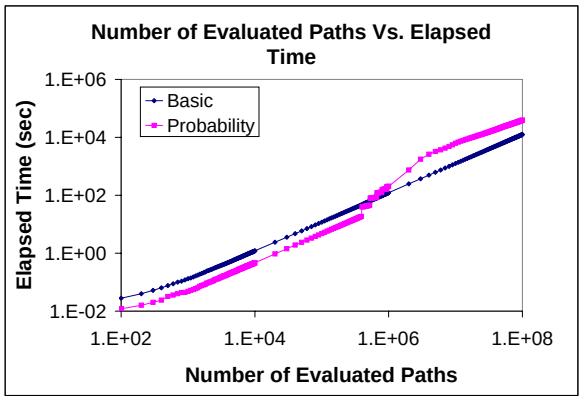
In terms of the smallest computed bound widths, the probability-based approach was able to attain the bound width of 6.05×10^{-5} when 1.0×10^{8} paths have been computed, which took 3.89×10^{4} seconds. The reward-based approach was able to attain the bound width of 1.24×10^{-4} when 4.4×10^{7} paths have been computed, which took 2.22×10^{4} seconds. The integrated approach was able to attain bounds that are as good as the probability-based approach, but it took much longer to compute the same amount of paths. That is shown by the divergent of the graphs for the approaches in Figure 4.13(c). Note that all three path-selection approaches was able to compute tighter bounds than the basic approach.



(a) Number of Paths Vs. Bound Width

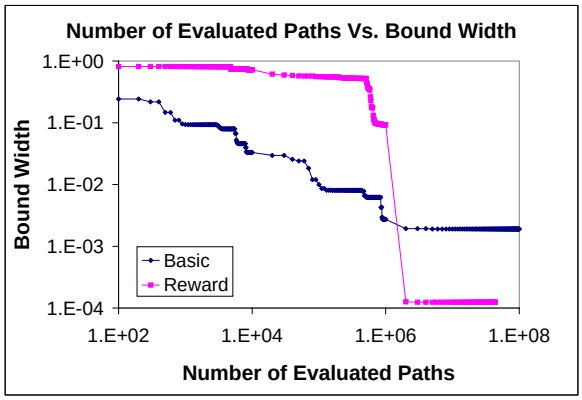


(b) Computation Time Vs. Bound Width

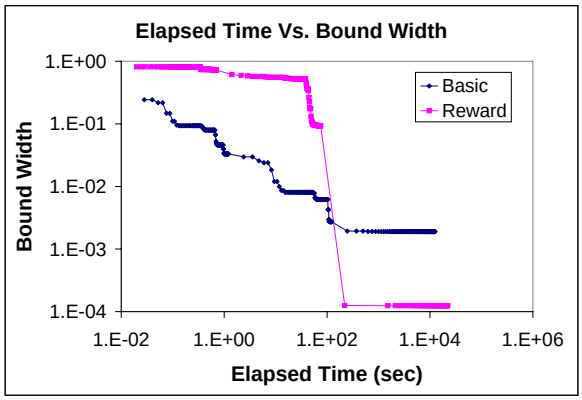


(c) Number of Paths Vs. Computation Time

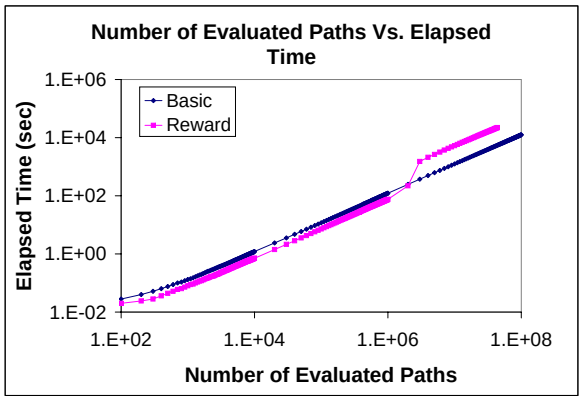
Figure 4.10: Availability model: Path selection according to probability values at time $t = 10.0$.



(a) Number of Paths Vs. Bound Width

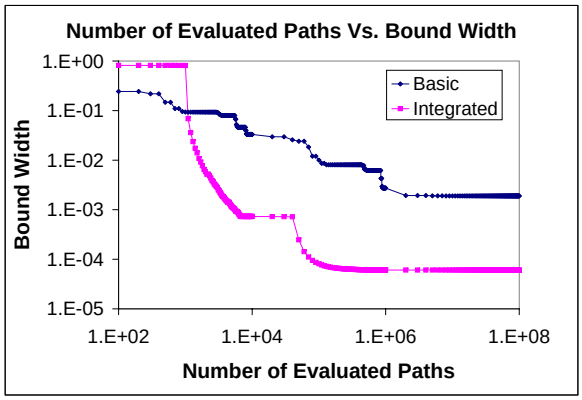


(b) Computation Time Vs. Bound Width

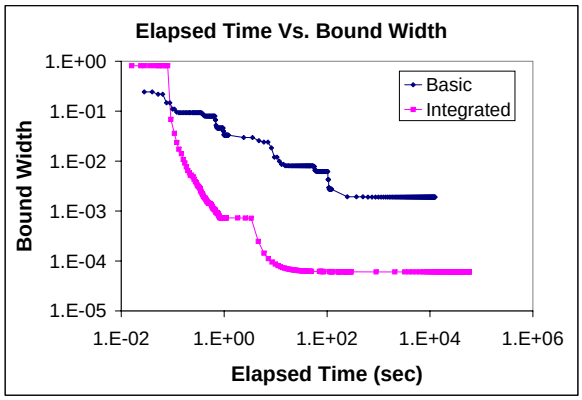


(c) Number of Paths Vs. Computation Time

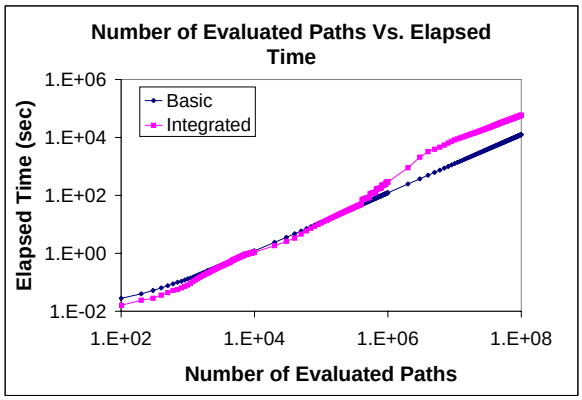
Figure 4.11: Availability model: Path selection according to reward values at time $t = 10.0$.



(a) Number of Paths Vs. Bound Width

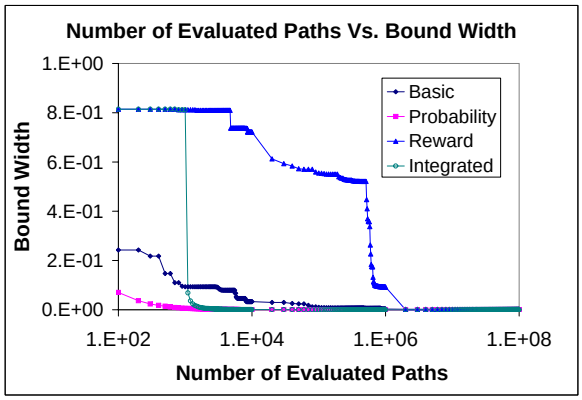


(b) Computation Time Vs. Bound Width

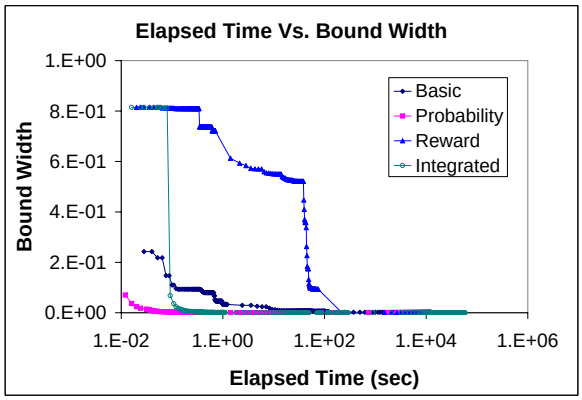


(c) Number of Paths Vs. Computation Time

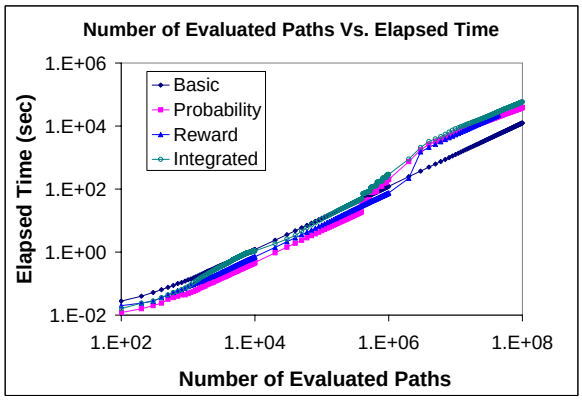
Figure 4.12: Availability model: Path selection by the integrated approach at time $t = 10.0$.



(a) Number of Paths Vs. Bound Width



(b) Computation Time Vs. Bound Width

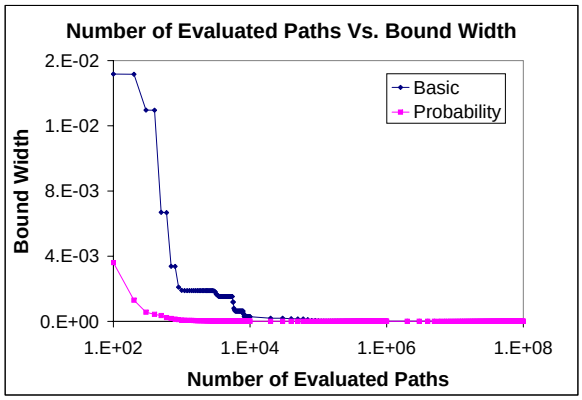


(c) Number of Paths Vs. Computation Time

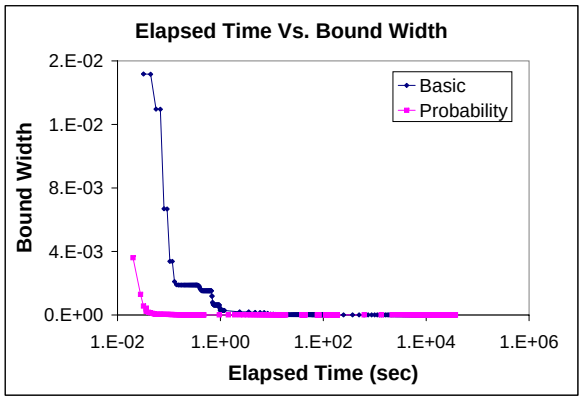
Figure 4.13: Availability model: Combined graphs of the three approaches at time $t = 10.0$.

The results for the evaluation of the availability model at transient time point 100.0 are shown in Figures 4.14 – 4.17. As shown in Figure 4.14, the probability-based selection approach again is able to compute tight bounds quickly. When the results of the bound widths computed at time 100.0 are compared to those computed at time 10.0, it is worth noting that the probability-based approach is able to compute tighter bounds faster for the model at time 100.0 than it can at time 10.0 (as compared between Figures 4.10 and 4.14). Those results are better than our expectation, because ordinarily any technique that is based on uniformization (on which the path-based approaches are based) takes more time to compute bounds for longer transient time than shorter ones. For example, through a number of experiments, we have noted that the approach described in Chapter 2, which does not use path selection, always computes larger widths for longer transient times. Thus, for some models, using the right path-selection approach can help to improve performance and to compute better bound width for longer transient time point.

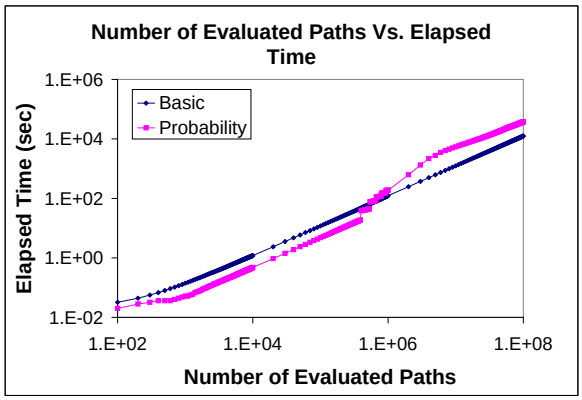
The results computed at transient time 10.0 show that path selection based on reward values is not an effective approach for solving the availability model. The results for transient time 100.0 show that fact more apparently. Note that in Figure 4.15(a), the graph for the results of the approach for selecting paths based on reward values is higher than that of the basic approach. A more detailed explanation for that fact is given in Section 4.4.4. Briefly, the main reason that the reward-based approach is not effective is the following. When a model is solved for longer transient time point, the results from evaluating longer paths become more relevant. Two important points, however, must be noted regarding longer paths in availability models: the path probability decreases quickly for longer paths and the set of possible path reward values is the same for long paths as it is for short paths. Thus, the path probability values play a more critical role than the path reward values in availability models when longer transient time is evaluated.



(a) Number of Paths Vs. Bound Width

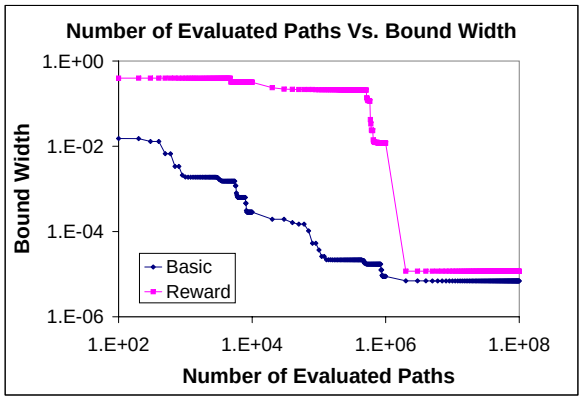


(b) Computation Time Vs. Bound Width

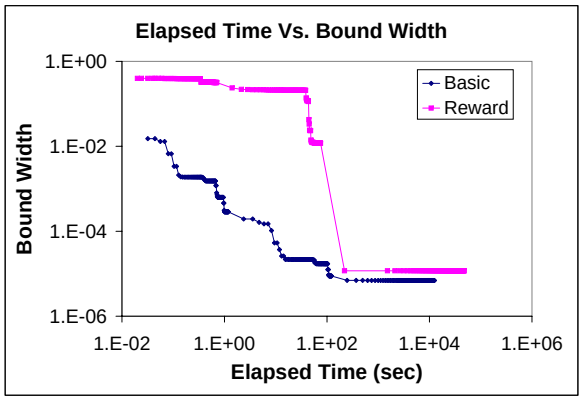


(c) Number of Paths Vs. Computation Time

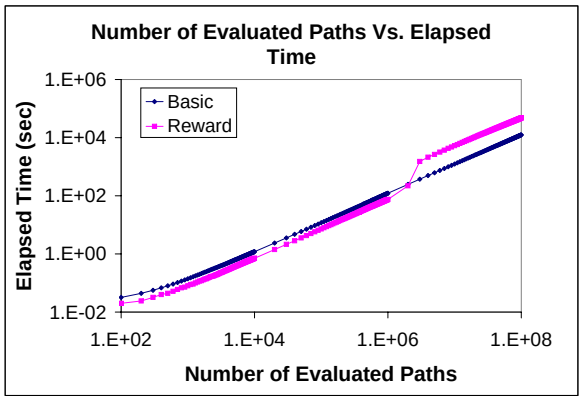
Figure 4.14: Availability model: Path selection according to probability values at time $t = 100.0$.



(a) Number of Paths Vs. Bound Width

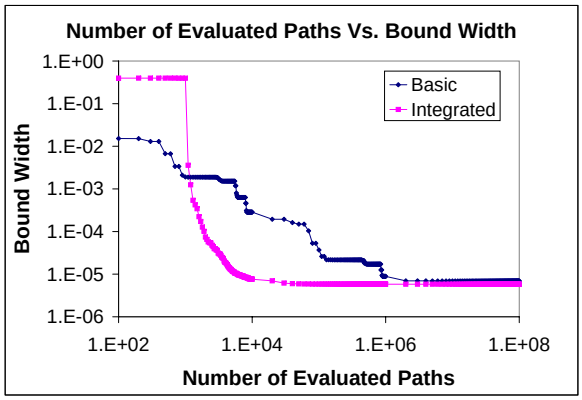


(b) Computation Time Vs. Bound Width

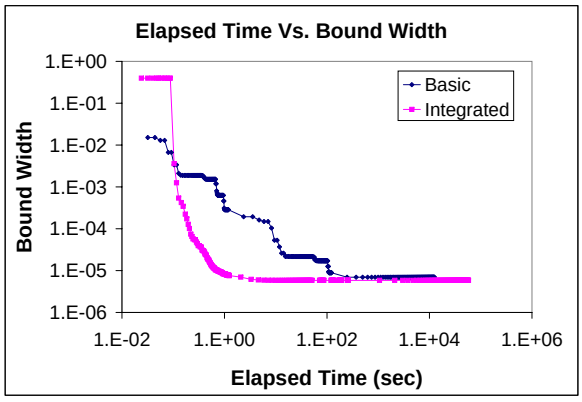


(c) Number of Paths Vs. Computation Time

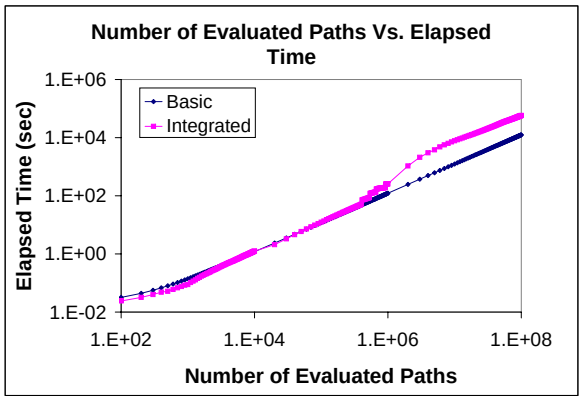
Figure 4.15: Availability model: Path selection according to reward values at time $t = 100.0$.



(a) Number of Paths Vs. Bound Width

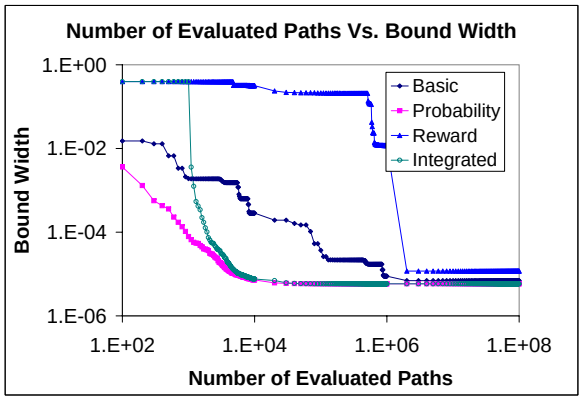


(b) Computation Time Vs. Bound Width

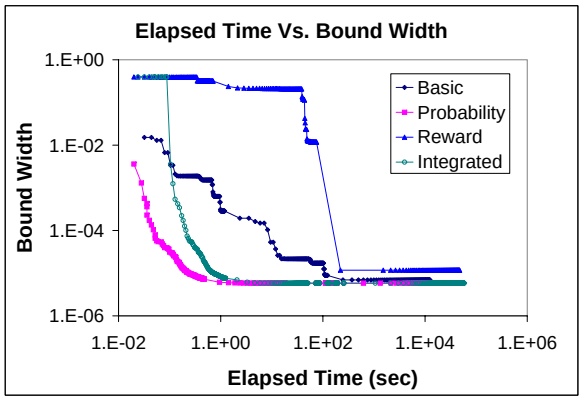


(c) Number of Paths Vs. Computation Time

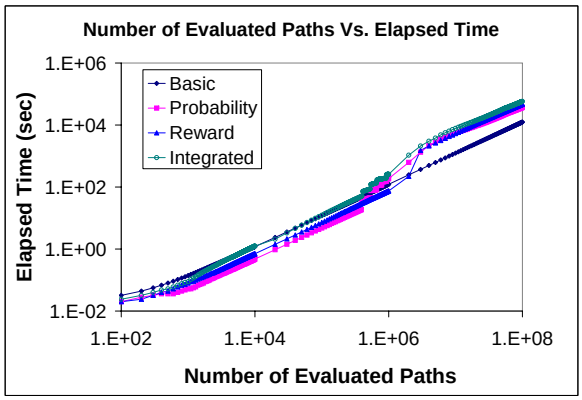
Figure 4.16: Availability model: Path selection by the integrated approach at time $t = 100.0$.



(a) Number of Paths Vs. Bound Width



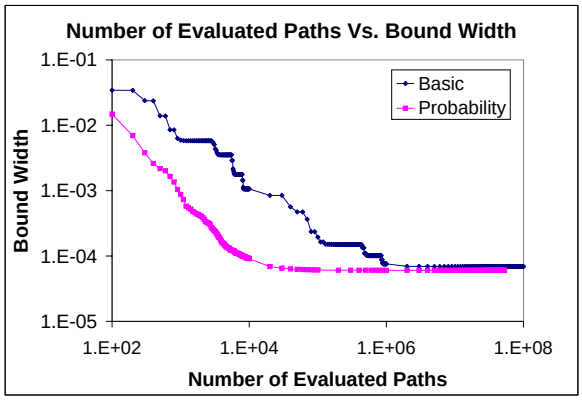
(b) Computation Time Vs. Bound Width



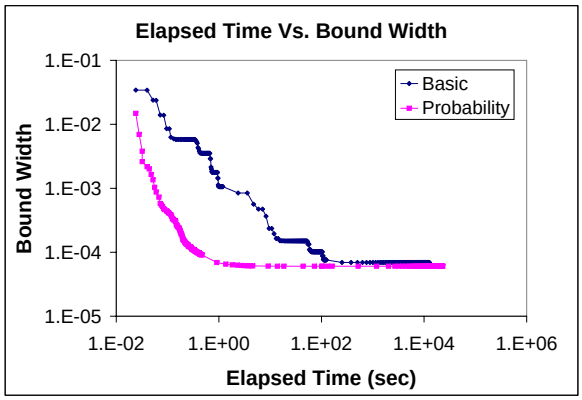
(c) Number of Paths Vs. Computation Time

Figure 4.17: Availability model: Combined graphs of the three approaches at time $t = 100.0$.

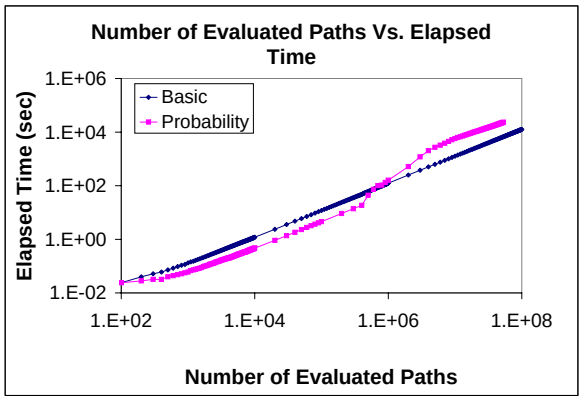
4.4.2 Reliability Model



(a) Number of Paths Vs. Bound Width

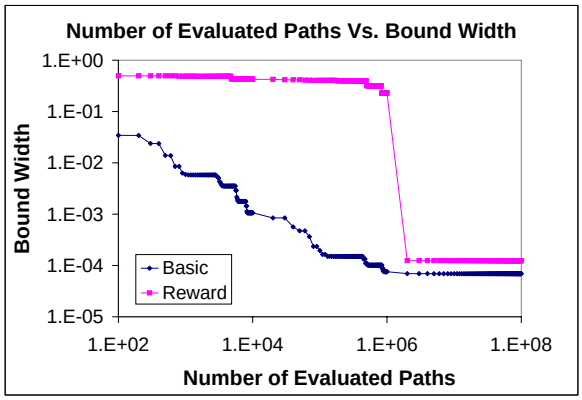


(b) Computation Time Vs. Bound Width

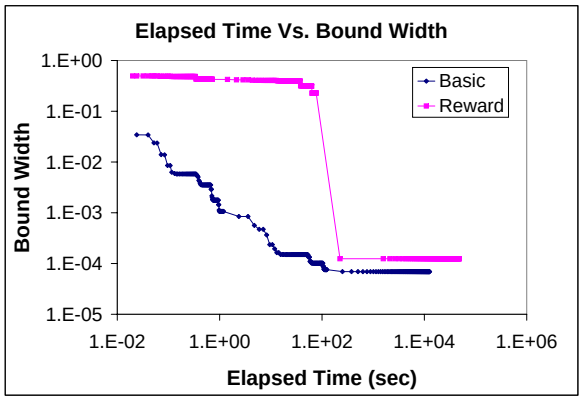


(c) Number of Paths Vs. Computation Time

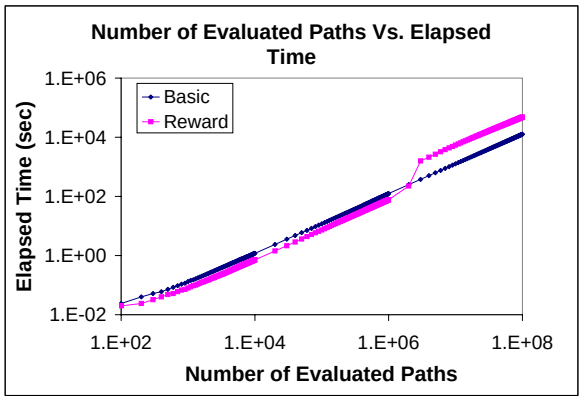
Figure 4.18: Reliability model: Path selection according to probability values at time $t = 10.0$.



(a) Number of Paths Vs. Bound Width

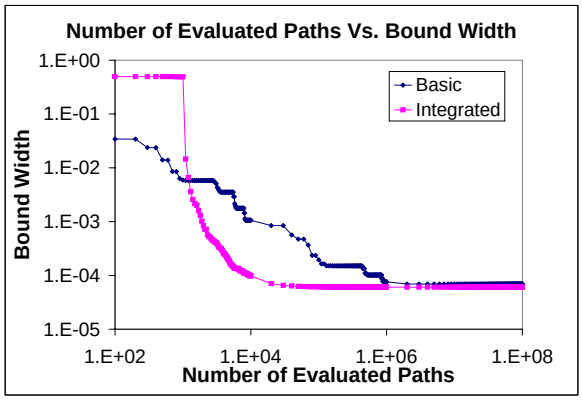


(b) Computation Time Vs. Bound Width

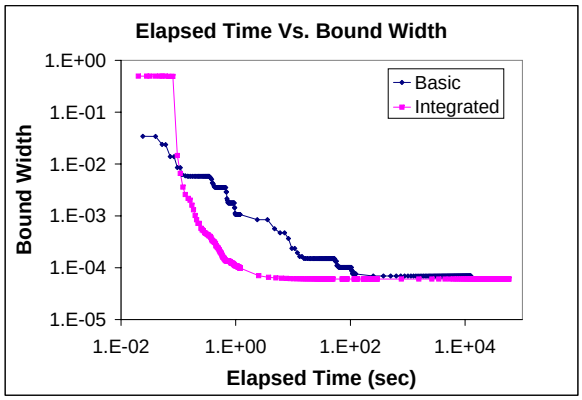


(c) Number of Paths Vs. Computation Time

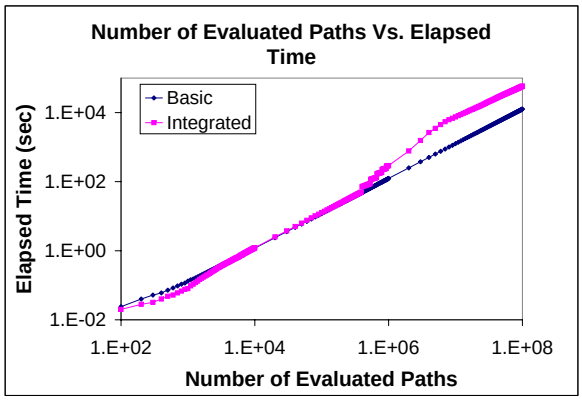
Figure 4.19: Reliability model: Path selection according to reward values at time $t = 10.0$.



(a) Number of Paths Vs. Bound Width

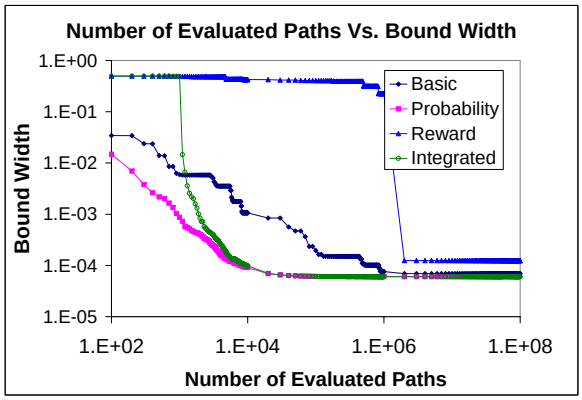


(b) Computation Time Vs. Bound Width

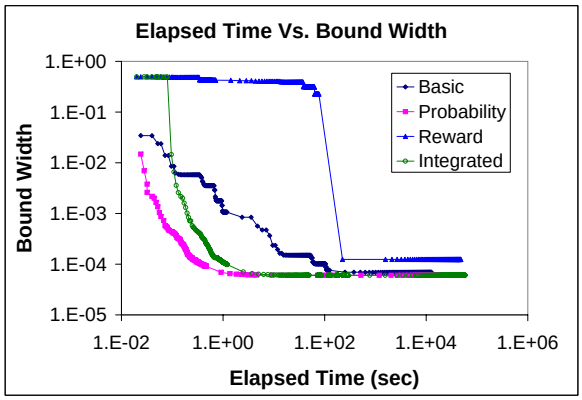


(c) Number of Paths Vs. Computation Time

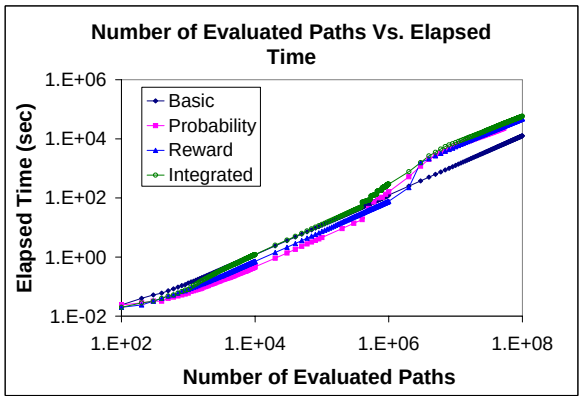
Figure 4.20: Reliability model: Path selection by the integrated approach at time $t = 10.0$.



(a) Number of Paths Vs. Bound Width

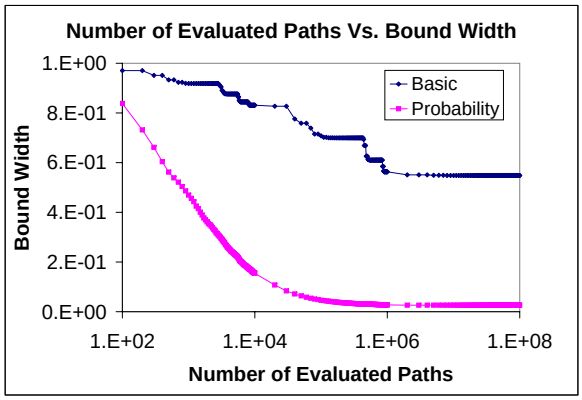


(b) Computation Time Vs. Bound Width

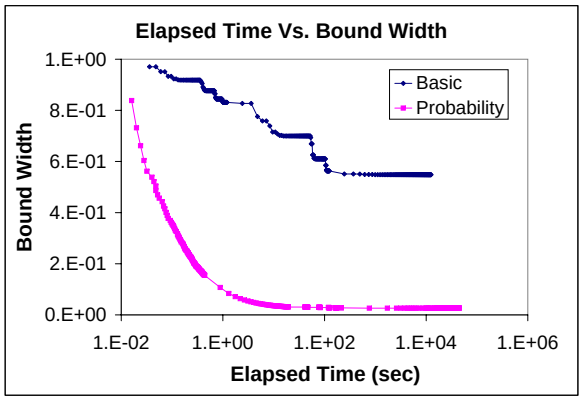


(c) Number of Paths Vs. Computation Time

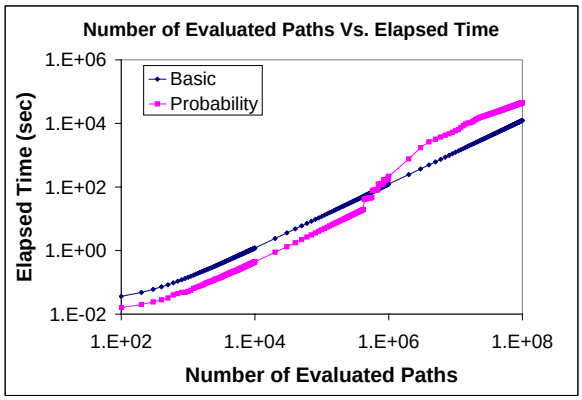
Figure 4.21: Reliability model: Combined graphs of the three approaches at time $t = 10.0$.



(a) Number of Paths Vs. Bound Width

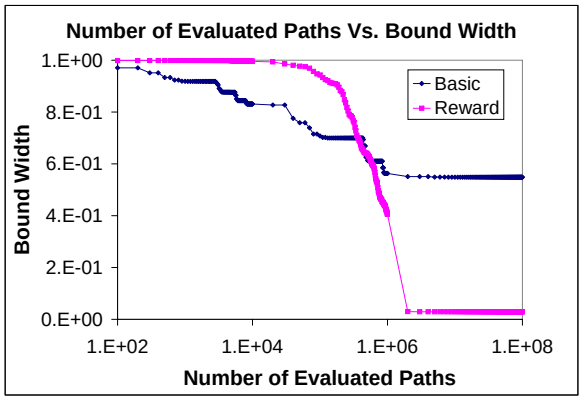


(b) Computation Time Vs. Bound Width

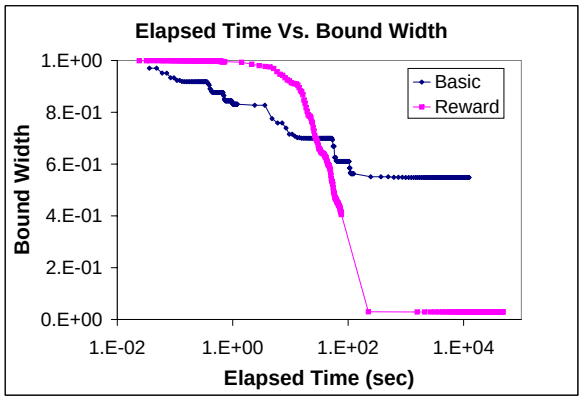


(c) Number of Paths Vs. Computation Time

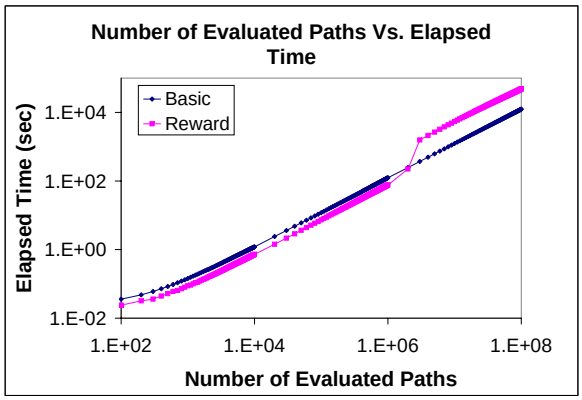
Figure 4.22: Reliability model: Path selection according to probability values at time $t = 100.0$.



(a) Number of Paths Vs. Bound Width

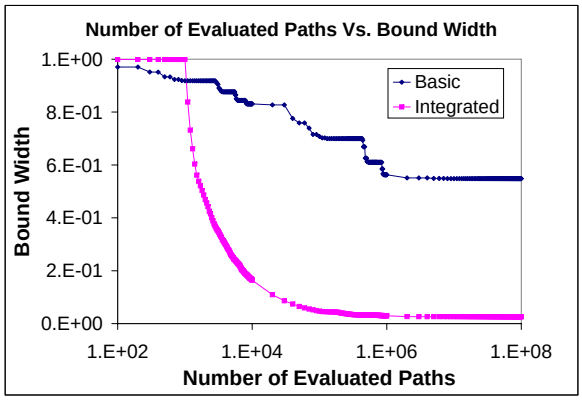


(b) Computation Time Vs. Bound Width

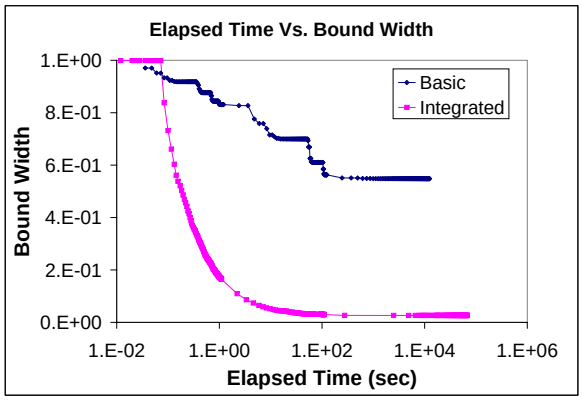


(c) Number of Paths Vs. Computation Time

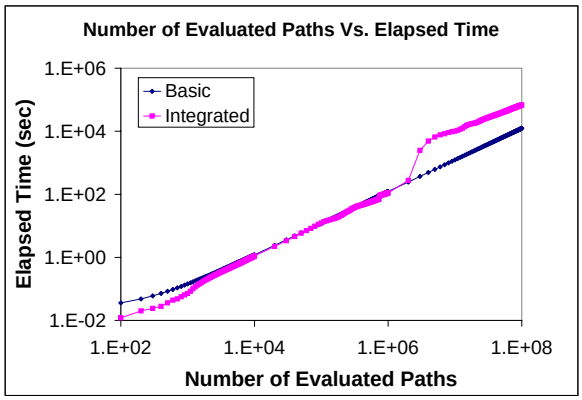
Figure 4.23: Reliability model: Path selection according to reward values at time $t = 100.0$.



(a) Number of Paths Vs. Bound Width

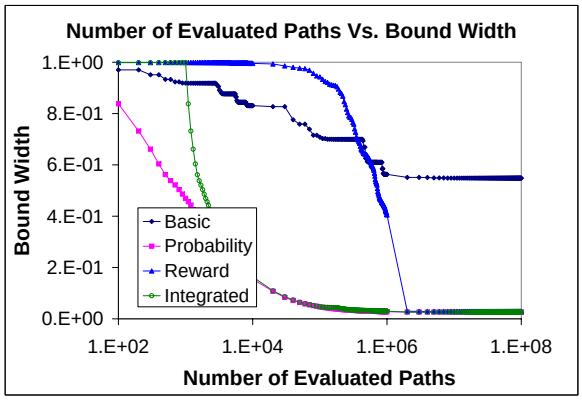


(b) Computation Time Vs. Bound Width

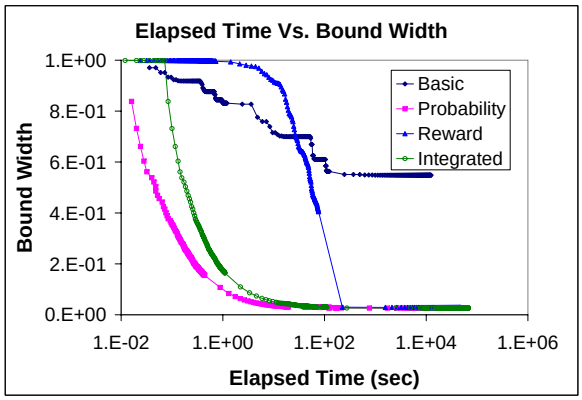


(c) Number of Paths Vs. Computation Time

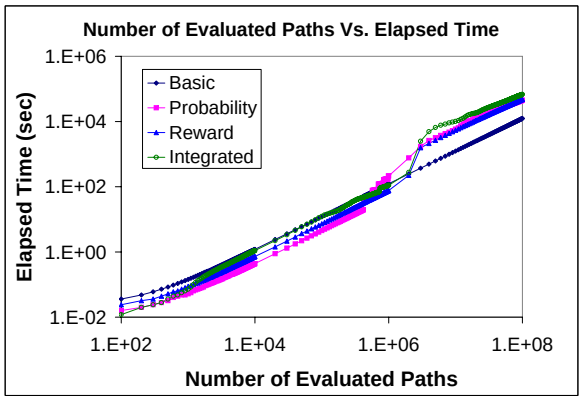
Figure 4.24: Reliability model: Path selection by the integrated approach at time $t = 100.0$.



(a) Number of Paths Vs. Bound Width



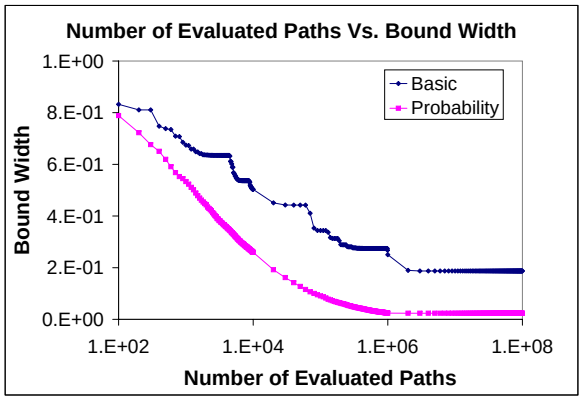
(b) Computation Time Vs. Bound Width



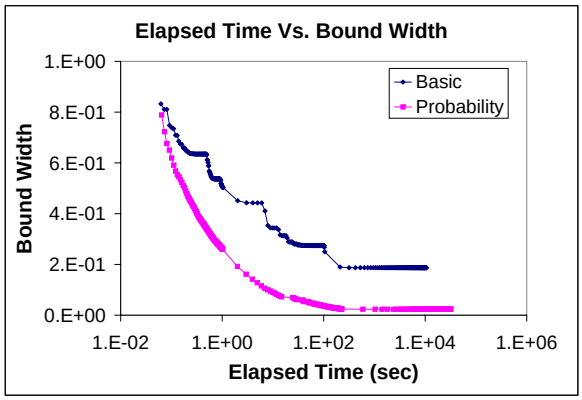
(c) Number of Paths Vs. Computation Time

Figure 4.25: Reliability model: Combined graphs of the three approaches at time $t = 100.0$.

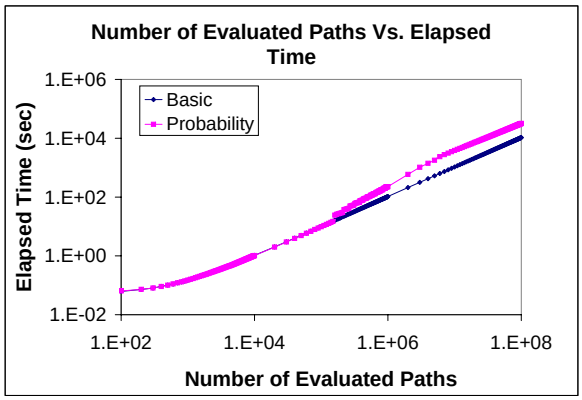
4.4.3 Performability Model



(a) Number of Paths Vs. Bound Width

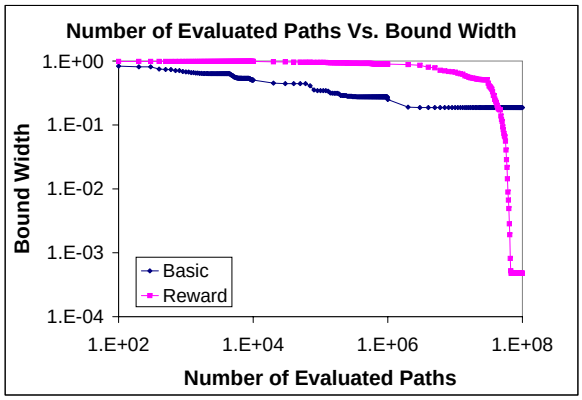


(b) Computation Time Vs. Bound Width

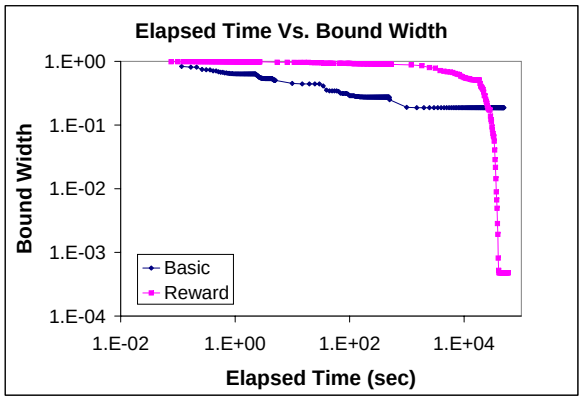


(c) Number of Paths Vs. Computation Time

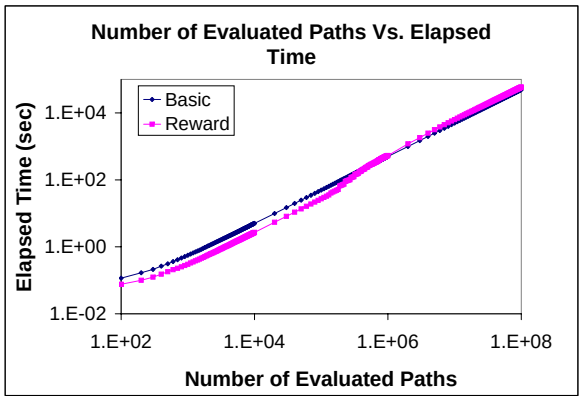
Figure 4.26: Performability model: Path selection according to probability values at time $t = 0.01$.



(a) Number of Paths Vs. Bound Width

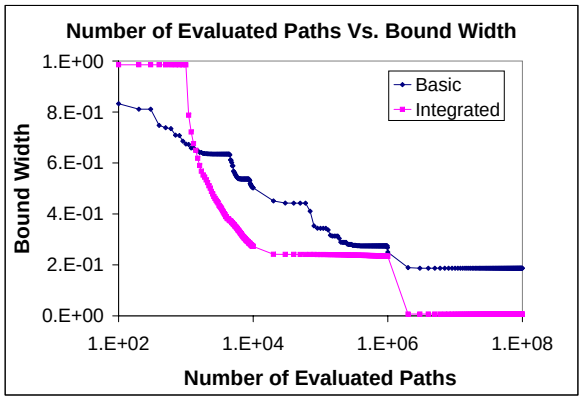


(b) Computation Time Vs. Bound Width

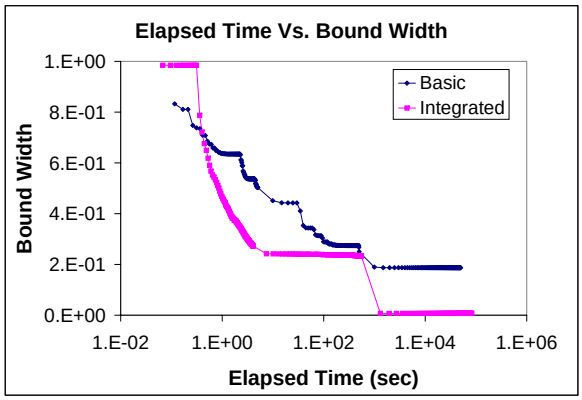


(c) Number of Paths Vs. Computation Time

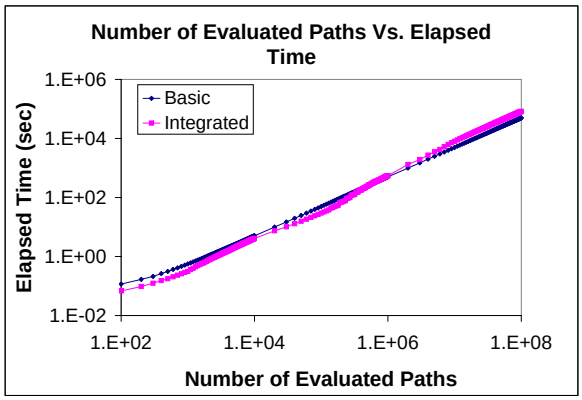
Figure 4.27: Performability model: Path selection according to reward values at time $t = 0.01$.



(a) Number of Paths Vs. Bound Width

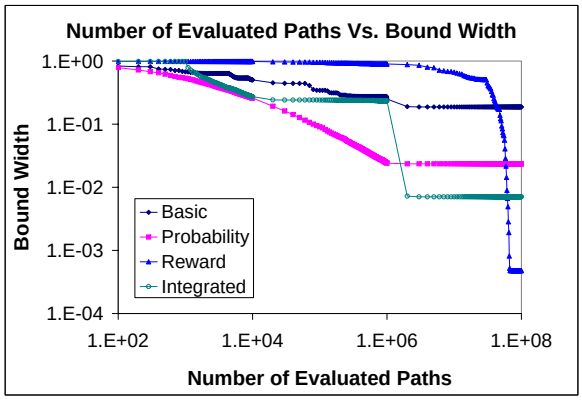


(b) Computation Time Vs. Bound Width

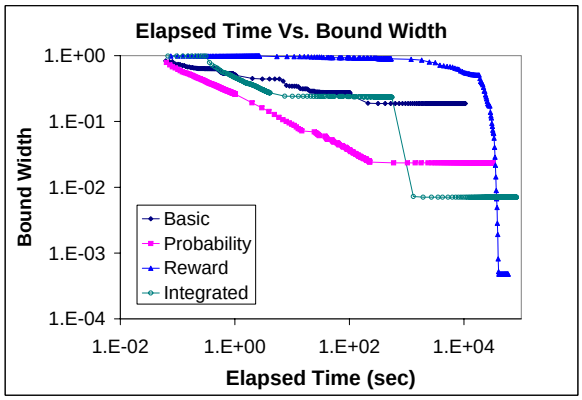


(c) Number of Paths Vs. Computation Time

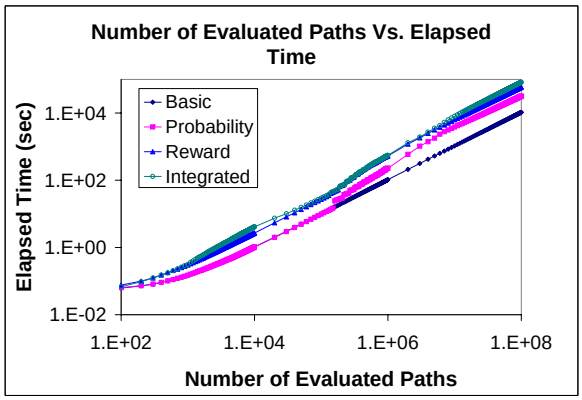
Figure 4.28: Performability model: Path selection by the integrated approach at time $t = 0.01$.



(a) Number of Paths Vs. Bound Width



(b) Computation Time Vs. Bound Width



(c) Number of Paths Vs. Computation Time

Figure 4.29: Performability model: Combined graphs of the three approaches at time $t = 0.01$.

Table 4.1: Reliability model: Results reproduced from Table 7 in [1]

Path Length	Bound Width	Canonical Paths	Potential Paths	Time (sec)
0	0.503415	0	0	0.03
1	0.155805	17	17	0.05
2	0.034174	232	306	0.04
3	0.005806	2871	5219	0.06
4	0.000845	35437	88740	0.56
5	0.000150	437967	1.5086e+06	6.54
6	0.000069	5.41298e+06	2.56462e+07	81.2
7	0.000061	6.68963e+07	4.35985e+08	997.02

4.4.4 Discussion of Results

Path selection seems to be an effective approach for computing tight bounds quickly. In particular, when we compare our current work to the work in [1], we see that our performance has improved by up to two orders of magnitude. For example, Table 4.1 is a reproduction of Table 7 in [1], in which we compute the bound widths from the original bound values. Note that computing the best bound width in Table 4.1 takes 997.02 seconds to evaluate 6.69×10^7 paths. Use of the probability-based path-selection approach to compute a comparable bound width, 6.10×10^{-5}, takes 4.56 seconds to evaluate 1.00×10^5 paths.

In general, for almost all of the experiments that we conducted, the path-selection approaches computed better bounds than the baseline approach. The two exceptions are the use of the reward-based approach to solve the availability model at time $t = 100.0$ (see Figure 4.15) and to solve the reliability model at time $t = 10.0$ (see Figure 4.19).

For the models that we evaluated, the path-selection approaches usually converge to the bounding solutions within several hundred seconds (note the flat lines in the (b) graphs of Figures 4.13, 4.17, 4.21, 4.25, and 4.29). That seems to indicate that the approaches are able to find many of the important paths that contribute toward tightening of the bounds after some amount of time. Beyond that amount of time, further evaluation does not seem to tighten the bounds significantly. There may be two ways to explain that phenomenon. It could be that the paths that were selected by the approaches earlier in the evaluation led to paths that contributed little toward tightening of the bounds. It is also possible that after some time, the remaining paths will become too long and have small probability values. Recall the probability curves in Figure 4.6. After some point, the curves monotonically decrease to small probability values. Tightening the

bounds by any significant amount after that point requires the evaluation of many paths.

Some models are more effectively solved using the probability-based path-selection approach than the reward-based approach, and vice versa. Using the right approach for a particular model is crucial in computing tight bounds. Using the less effective approach could cause some paths to be discarded earlier (by the memory management module). That could result in important successor paths not being evaluated and resulting in loose bounds. For example, in Figure 4.21(a), the probability-based path-selection approach is more effective, since it computes tighter bounds than those computed by the reward-based approach when 1.0×10^8 paths have been evaluated. For the performability model, however, the reward-based approach is more effective when 1.0×10^8 paths have been evaluated as shown by Figure 4.29(a).

The integrated approach seems to be a reasonably good choice when it is not simple to determine whether the reward-based or probability-based approach is the more effective one for solving a model. Because it can switch between the two approaches dynamically based on runtime conditions, it usually obtains solutions that are close to the best ones. For example, observe in the (b) graphs of Figures 4.13, 4.17, 4.21, and 4.25 that the integrated approach switches a number of times between the probability-based and reward-based approaches and converges toward the best solutions fairly quickly. It is susceptible, however, to several conditions that may cause it to be less effective. Initially, it has to evaluate one set of paths using the probability-based approach and another set using the reward-based approach to determine which approach it should switch to next. Depending on which set it evaluates first, the approach may be led to a worse sequence of switches from the beginning. Furthermore, there is a time lag between the point when a set of paths is evaluated to calculate a slope and the point when a switch is made based on that slope. After the switch, the chosen approach could turn out to be the less effective approach for the next set of paths to be evaluated. That situation could lead to a "vicious cycle" and could cause some important paths to be deleted in the process. Figure 4.29(b) shows that the integrated approach worked rather well in solving the performability model, but did not do as well as the reward-based approach.

4.5 Summary

The structured path-based approach, described in Chapter 2, provides a means for evaluating very large Markov models by evaluating their evolutionary paths. To obtain tight bounds, the

approach has to evaluate many paths. That is inefficient and can be time-consuming.

In this chapter, we present three path-selection approaches for solving Markov models quickly. Two of the approaches select paths according to their reward and probability values. The third one dynamically switches between the two selection criteria according to the slopes of the bounds. The approaches are augmented with a memory management module to help them avoid thrashing the memory system when a large amount of memory is used.

We have shown that the path-selection approaches can improve performance up to two orders of magnitude relative to the structured path-based approach described in 2. In most experiments, we have shown that they can compute tighter bounds faster than the baseline approach (which is augmented with the memory management module and evaluates paths in FIFO order).

CHAPTER 5

PATH-BASED SIMULATION

5.1 Introduction and Motivation

Model-based evaluation is an effective means to gain insight into the behavior of many computing systems. It can be used during many stages of system development, from conceptual design to prototype evaluation to actual-system operation and maintenance. At all these stages, models of performance, availability, and reliability may be used to predict the operating characteristics of systems. Frequently, models can be expressed as continuous-time Markov chains (CTMCs). However, it is generally necessary to model a system at a fairly high level of detail in order to obtain results efficiently. As the amount of detail and the level of complexity in a model increase, computing solutions by numerical methods becomes increasingly difficult due to the well-known state-space explosion problem.

Simulation is an alternate solution approach that is usually used for analyzing very large models. It has no explicit state-space storage constraint because it analyzes a model directly without the use of the underlying state space. It does this by executing a representative set of trajectories over which the model may evolve over time. The trajectories are chosen based on their relative likelihood of occurrence in the model, and the solution of the model is estimated using a statistical analysis of the chosen trajectories.

In many simulation approaches, the trajectories are defined at the level of individual states and transitions. At this low level of representation, it is necessary to keep track of many state variables of a model to determine which transitions are enabled and which events may occur next. Furthermore, even though multiple trajectories can be evaluated in parallel on a multiprocessor machine, each one must still be evaluated serially on a uniprocessor machine. That constraint limits the number of trajectories that can be evaluated in a given amount of time. When a high confidence level is required in a solution, a large number of trajectories have to be evaluated.

Thus, solving a model for solutions with high confidence levels is a time-consuming task.

In this chapter, we present a new approach for simulating continuous-time discrete-state Markov models. The approach extends existing simulation and path-based analytical approaches by simulating models at the component level. Instead of representing the states of a model explicitly, as is traditionally done in many other simulation methods, our approach represents them with a set of probability vectors corresponding to the probabilities of being in particular states within each component of a model. By traversing "trajectories" at this higher level of abstraction, the approach improves the efficiency and quality of a solution relative to a solution computed using traditional discrete-event simulation or analytical path-based methods.

The presentation of the materials in this chapter proceeds as follows. In Section 5.2, we briefly explain how to represent a model at the component level. Then, in Section 5.3, we present a path-based simulation approach for executing a trajectory at the component level. Next, in Section 5.4, we illustrate how the approach can be used for evaluating the availability and reliability of a distributed information service system. Additionally, we present experimental results to compare the performance and effectiveness of the approach against the analytical path-based approach presented in [1]. Finally, we conclude with a summary and a discussion of future work in Section 5.5.

5.2 Component-Level Analysis of Models

Many large models, like their physical system counterparts, are constructed from smaller logical submodels or components. A large model is composed from the submodels either by means of *state sharing*, whereby the submodels coordinate their transitions through shared states (e.g., [43]), or by means of *action synchronization*, whereby they coordinate through shared events. For our approach, we consider the latter.

Suppose a given model has state space $\mathcal{RS} = \{0, \ldots, n-1\}$ (such that $|\mathcal{RS}| = n$) and is decomposable into J components, which are numbered from 1 to J. In the sequel, we use parenthesized superscripts and subscripts, i,$j \in \{1, \ldots, J\}$, to denote particular components, unless the context is clear (in which case, we do not use parentheses). Let component i have state space $\mathcal{RS}_i = \{0, \ldots, n_i - 1\}$. Then the state space of the entire model can be composed from the state spaces of the components such that the relation $\mathcal{RS} \subseteq \times_{i=1}^{J} \mathcal{RS}_i$ holds true.

In many simulation methods, a state of a component is represented by a vector of its state

variables; the state of the model is represented by a vector of component states. The next events that *may* occur are determined by observing the values of the state variables of the current state. When all conditions are satisfied to enable an event, it is scheduled and put on a list of enabled events. Then, one event from the list is chosen to occur. After the occurrence of an event, the affected state variables are updated, and a transition to the next state takes place. Afterward, events previously scheduled on the list may become disabled and are removed from the list.

The costs of determining the next events, managing the event lists, and updating states and state variables can (when all state holding times are exponentially distributed) be eliminated or lessened by keeping track of the probabilities that a component will occupy individual states instead of keeping track of the states that the component may occupy. Essentially, instead of using a vector of state variables for each component, a vector of state probabilities is used to keep track of the probabilities that the component will occupy the states in its state space. The *state probability vector* of the whole model can be represented in a similar manner and can be computed from the state probability vectors of the components. If this representation is used, it is not necessary to determine which next events are enabled; it is not necessary to manage an event list; and updating states involves only a (more efficient) vector-matrix operation.

More formally, define $\mathbf{p}_0^{(i)}$ to be the initial state probability (row) vector of component i and $\mathbf{p}^{(i)}$ to be the vector after some number of events that affect i have occurred. The events that affect i can be classified as either *local* to i or *global*. The local events are those occurring within component i. The global events are those associated with the synchronization activities that affect multiple components in the whole model. Each synchronization activity can be further classified as either a *true synchronization event*, in which *all* involved components take a synchronized transition, or a *disabled synchronization event*, in which *some* involved components cannot take the synchronized transition because their present states disallow the transition.

The effects of these events on the components can be described by the *probability transition matrices* [5]. Each matrix describes how a component transitions among its states and the probabilities that those transitions will be caused by the event. Let us define the following matrices corresponding to these types of events for component i: $\mathbf{P}_l^{(i)}$ is the local transition matrix, $\mathbf{E}_t^{(i)}$ is the synchronized transition matrix for synchronization event t, and $\overline{\mathbf{E}}_t^{(i)}$ is the disabled synchronized transition matrix for synchronization event t. $\mathbf{D}_t^{(i)} = \text{diag}(\mathbf{E}_t^{(i)}\mathbf{e}^T)$, where $\mathbf{e}$ is an n_i-dimensional vector of ones, is simply the diagonal matrix of $\mathbf{E}_t^{(i)}$. We summarize these

matrices notationally as follows:

$$\mathbf{\Phi}^{(i)}_{\boldsymbol{\pi}(k)} = \begin{cases} \mathbf{P}^{(i)}_{l} & \text{if } \boldsymbol{\pi}(k) = l_i \\ \mathbf{E}^{(i)}_{t} & \text{if } \boldsymbol{\pi}(k) = t \text{ for } t \in \mathcal{T}_S \\ \mathbf{D}^{(i)}_{t} & \text{if } \boldsymbol{\pi}(k) = \bar{t}_j \text{ for } i < j \text{ and } t \in \mathcal{T}_S \\ \overline{\mathbf{E}}^{(i)}_{t} & \text{if } \boldsymbol{\pi}(k) = \bar{t}_i \text{ for } t \in \mathcal{T}_S \\ \mathbf{I}_{n_i} & \text{otherwise} \end{cases} .$$

By using these vectors and matrices, the effect of a series of events on a particular component can be computed simply by a series of products of the corresponding component vector and matrices.

It is possible to compute various measures of interest from a model by using the Markov reward model [54] to specify how reward is computed when the model visits some states. For computing rate reward, define the column vector $\mathbf{r}^{(i)}$ to be the reward vector for component i. $\mathbf{r}^{(i)}$ is an n_i-dimensional vector whose entries specify the reward associated with the corresponding states. Further, let $\boldsymbol{\pi}^{(i)}$ be a sequence of events that affect i. $\boldsymbol{\pi}^{(i)}$ can also be considered as a *set of paths* over which i evolves over time, and it is an abstraction of the state-level trajectory. Then the product of $\mathbf{p}^{(i)}_0$ and the matrices corresponding to the events in $\boldsymbol{\pi}^{(i)}$ yields the state probability vector, $\mathbf{p}^{(i)}[\boldsymbol{\pi}^{(i)}]$, after the occurrence of $\boldsymbol{\pi}^{(i)}$. The reward from component i over this path is computed as

$$R^{(i)}(\boldsymbol{\pi}^{(i)}) = \mathbf{p}^{(i)}[\boldsymbol{\pi}^{(i)}] \cdot \mathbf{r}^{(i)}. \tag{5.1}$$

At a global model level, let $\boldsymbol{\pi}$ be a path of events over which the model evolves. Observe that $\boldsymbol{\pi}$ is a splicing of $\boldsymbol{\pi}^{(i)}$ for $i = 1 \ldots J$. For a measure specified by a tensor product [5] of the component reward vectors, the reward over $\boldsymbol{\pi}$ is computed as

$$R(\boldsymbol{\pi}) = \prod_{k=1}^{J} R^{(i)}(\boldsymbol{\pi}^{(i)}). \tag{5.2}$$

The probability of the path $\boldsymbol{\pi}$ is given by

$$Prob(\boldsymbol{\pi}) = \prod_{k=1}^{|\boldsymbol{\pi}|} \frac{\lambda_{\boldsymbol{\pi}(k)}}{\Lambda}, \tag{5.3}$$

where $\lambda_{\boldsymbol{\pi}(k)}$ is the rate of occurrence of the event corresponding to $\boldsymbol{\pi}(k)$ and Λ is the sum of the

rates of enabled transitions in the model. Let $\beta(\Lambda s, k)$ denote the Poisson density describing the probability of having k occurrences of some events at rate Λs. Then the expected reward of the model at time s can be computed by

$$E[R_s] = \sum_{k=0}^{\infty} \beta(\Lambda s, k) \sum_{\boldsymbol{\pi} \in \mathcal{P}^k} Prob(\boldsymbol{\pi}) R(\boldsymbol{\pi}). \tag{5.4}$$

Equation (5.4) states that the expected instantaneous reward is just the weighted sum of the path rewards over all possible paths, where the weights are the probabilities of the paths.

Note that computation of the exact solution requires evaluation of infinitely many paths as shown in Equation (5.4). Guaranteed lower and upper bounds on the exact solution can be computed by evaluating a finite number of paths; the more paths that are evaluated, the tighter the bounds will be (as shown in [1]). In general, to get useful bounds for practical models, a large number of paths must be evaluated. In the next section, we present a new path-based simulation approach that evaluates many fewer paths than the analytical path-based bounding approach of [1], yet it can efficiently compute solutions with high confidence. The approach is not only more effective than existing ones, it is also more efficient.

5.3 The Path-Based Simulation Approach

This section presents the component-level path-based simulation approach. It is more efficient than traditional simulation methods because it is partially based on numerical analysis and it does not manage any event list. It is also often faster than the analytical path-based approach because it can estimate a solution by evaluating fewer paths. After presenting the approach, we use it to evaluate one model for the availability measure and another model for the reliability measure. We then compare the performance and effectiveness of the approach in computing these solutions against the analytical path-based bounding approach.

The basic idea behind the path-based simulation approach is that it is possible to estimate the solution of Equation (5.4) by sampling a set of random paths. Formally, let $\mathcal{SP}$ be a *multiset* of (sets of) paths from which samples will be drawn. Note that $\mathcal{SP}$ is a (small but representative) subset of the set of all possible paths, and the same path may be drawn multiple times from it. A

path $\boldsymbol{\pi} \in \mathcal{SP}$ must be chosen with probability

$$\beta(\Lambda s, |\boldsymbol{\pi}|) \cdot Prob(\boldsymbol{\pi}). \tag{5.5}$$

Then the unbiased estimators for the mean and variance can be computed by

$$\begin{aligned} \hat{R}(\mathcal{SP}) &= \frac{1}{|\mathcal{SP}|} \sum_{\boldsymbol{\pi} \in \mathcal{SP}} R(\boldsymbol{\pi}) \\ \hat{S}^2(\mathcal{SP}) &= \frac{1}{|\mathcal{SP}| - 1} \sum_{\boldsymbol{\pi} \in \mathcal{SP}} \left(R(\boldsymbol{\pi}) - \hat{R}(\mathcal{SP}) \right)^2. \end{aligned} \tag{5.6}$$

In order to keep the estimators unbiased, it is necessary to exercise caution in choosing random paths according to their probabilities of occurrence as given in Equation (5.5). It is possible to choose a path randomly according to the probability by first drawing a random variate from the Poisson distribution that has rate Λs. For an efficient realization, we use the work of [55] in our implementation. A Poisson random variate is used to choose the length of each sampled (abstract) path. Next, the events making up the path must be chosen according to their probabilities. Note that they cannot be statically determined as in Equation (5.3), but instead they must be computed dynamically as the model evolves according to the path $\boldsymbol{\pi}$. That is, the probabilities of the events to be chosen to form the path $\boldsymbol{\pi}$ are computed according to the states that the model occupies. These probabilities are computed by

$$ProbT(\boldsymbol{\pi}(k)) = \frac{\lambda_{\boldsymbol{\pi}(k)}}{\Lambda} \prod_{j=1}^{J} \mathbf{p}^{(j)} \cdot \boldsymbol{\Phi}^{(j)}_{\boldsymbol{\pi}(k)} \cdot \mathbf{e}^T_{n_j}, \tag{5.7}$$

where $\boldsymbol{\pi}(k) \in \{l_j, t, \bar{t}_j\}$ for $j = 1 \ldots J$ and $t \in \mathcal{T}_S$. Equation (5.7) is simple to compute and can be optimized so as to require only a scalar product. Thus, $ProbT(\boldsymbol{\pi}(k))$ forms a distribution over the set of possible next events.

Using Equation (5.6), an approximate $100(1-\alpha)\%$ $(0 < \alpha < 1)$ confidence interval for the mean is computed by

$$\hat{R}(\mathcal{SP}) \pm t_{|\mathcal{SP}|-1,1-\alpha/2} \sqrt{\frac{\hat{S}^2(\mathcal{SP})}{|\mathcal{SP}|}}, \tag{5.8}$$

where $t_{|\mathcal{SP}|-1,1-\alpha/2}$ is the *Student t-distribution* having $|\mathcal{SP}| - 1$ degrees of freedom.

5.4 Evaluation of the Approach

We evaluate our approach by studying its performance in analyzing a model of a distributed information service system adapted from the model in [48] and another model of a media multicast system. The first model describes the propagation of faults across the components of the system. We augment the original model with synchronized transitions among the components as a means for modeling how faults are propagated. In addition, we increase the number of front-end modules in order to model the occurrence of a fault only when a majority of the modules are corrupted. We also model double redundancy in the processing units by adding an additional module for every module in the original processing units.

We also evaluate the approach by analyzing a media multicast system. That model is different from the distributed information service system in that it has a higher degree of interconnection among the components and it has a much larger state space. After describing the model, we present the experimental results and discuss the performance and accuracy of the approach.

5.4.1 Model Description: Double-Redundant Distributed Information Service System

The information service system consists of six front-end modules that interact with four processing units. Each processing unit consists of redundant components, including two processors, two switches, two memory units, and two databases. Each of the components has its own repair facility. The components all go through the cycle of *Working*, *Corrupted*, *Failed*, and *Repaired*. For the reliability model, the repair facility is excluded. The *stochastic activity network* (SAN) models for evaluating the availability and reliability measures of the system are shown in Figures 3.1 and 3.2, respectively. Many of the parameter values for the models are given in Chapter 2.

The model has five components: the front end and 4 processing units. Each unit contains two processors, two switches, two memory units, and two databases. We analyze the model to compute the system reliability and availability. We compute the reliability measure at transient time point 1.0 and the availability measure at transient time point 0.1 when all components in the model are in the working state.

5.4.2 Model Description: Media Multicast System

The model of the media multicast system is shown in Figure 3.3. It consists of a source and six clients. The source generates media frames that are then multicasted to the clients. Buffers in the system overflow when the source transmits frames too quickly or when the client buffers are not large enough. When the source or clients become faulty, their buffers are flushed. Thus, by varying buffer size, transmission rates, and error rate, we can study measures such as the probabilities of experiencing buffer overflow or buffer flushing in the system. The model has approximately 1.9×10^{20} states.

5.4.3 Experimental Results

We conducted all of our experiments on a workstation that had the AMD Athlon XP 2700+ processor running at 2.17 GHz with 512 MB of RAM. The operating system was Red Hat Linux 9.0 with mounted file systems. We compiled our implementation using the compiler g++ 3.3 with optimization flag -O3 only.

In order to evaluate the quality of the simulation results, we initially analyzed the model for the guaranteed lower and upper bounds on the exact solutions using the approach in Chapter 2. For the point availability measure, the bounds are $[9.786854 \times 10^{-1},\ 9.985875 \times 10^{-1}]$ and take 6135.23 seconds to compute. For the reliability measure, the bounds are $[9.836687 \times 10^{-1},\ 9.852736 \times 10^{-1}]$ and take 487.09 seconds to compute.

The path-based simulation results are $9.9839562118 \times 10^{-1} \pm 5.3630001139 \times 10^{-5}$ for the point availability (computed in 12.738 seconds) and $9.8353716323 \times 10^{-1} \pm 9.9889749668 \times 10^{-4}$ for reliability (computed in 10.960 seconds).

The simulation results for point availability were obtained at a 95% confidence level and $6.0E-5$ absolute confidence interval width. The results for reliability were obtained at a 95% confidence level and $1.0E-3$ absolute confidence interval width. For all simulation experiments, our simulator terminated only when the computed confidence intervals fell below these specified ones.

Figures 5.1(a) and (b) present the above results in a graphical format for easier comparison of the performance and quality of the solutions obtained using the analytical path-based approach and the path-based simulation approach. In the figures, the bars are not lined up vertically so that the error bars can be distinguished clearly. The vertical axes show the measures that were

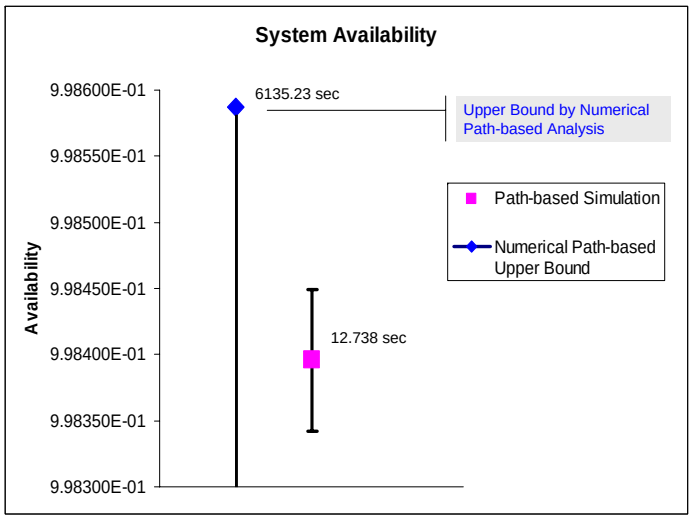
computed, and the exact solution lies somewhere between the upper and lower bounds computed by the analytical path-based approach. The number next to each bar indicates the time, in seconds, taken by the corresponding approach.

As shown by the intersection of the error bars for the path-based simulation approach and the bounds, the path-based simulator is capable of estimating the true solutions quite accurately. In Figure 5.1(a), note that the analytical approach took more than 6100 seconds to compute the bounds, and yet they are not tight. Thus, given the quality of the estimated solutions, the path-based simulator takes much less time than the analytical approach for both models.

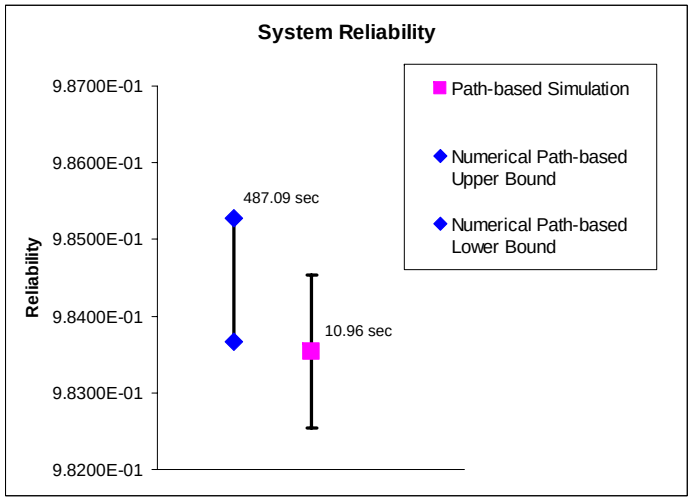
Figure 5.2 shows similar results for the model of the media multicast system. Figure 5.2(a) shows the result for the probability of experiencing buffer overflow, and Figure 5.2(b) shows the probability of experiencing buffer flushing. Again, the simulation approach can compute tight confidence intervals quickly. For that model, performance is improved up to three orders of magnitude through the use of the path-based simulation approach.

5.5 Summary

In this chapter, we presented a novel path-based simulation approach for analyzing very large Markov models. The approach raises the abstraction level of trajectory analysis to the higher level of component analysis. In doing so, we gain two benefits: first, sets of trajectories are simultaneously evaluated at the same time to improve the quality of the solution for a given number of traversals, and second, the approach works more efficiently by not having to manage event lists. As shown in Figures 5.1 and 5.2, the solutions estimated by the path-based simulation approach are quite accurate as compared with the guaranteed bounds on the true solutions. In addition, the simulation approach is much faster by several orders of magnitude than the analytical approach, making it quite usable for analyzing large and realistic models.

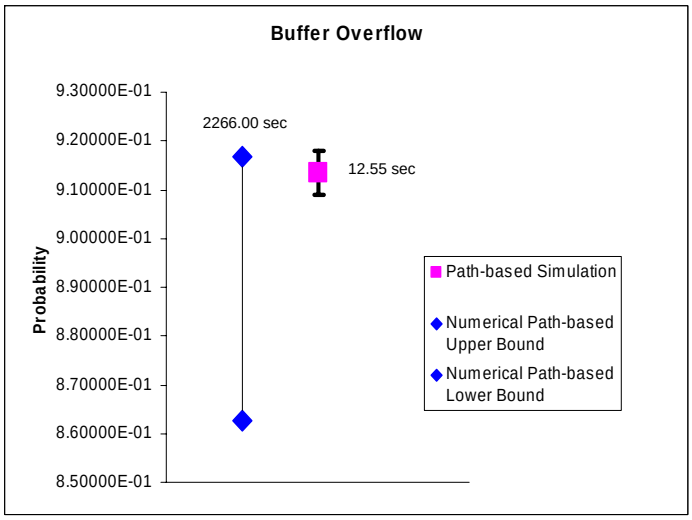


(a) Availability

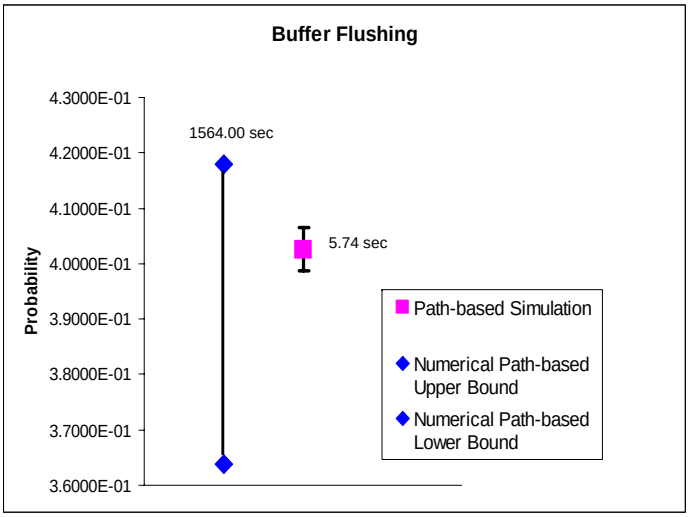


(b) Reliability

Figure 5.1: Comparison of the performance and quality of the solutions obtained using the analytical path-based approach and the path-based simulation approach on the distributed information service system.



(a) Buffer Overflow



(b) Buffer Flushing

Figure 5.2: Comparison of the performance and quality of the solutions obtained using the analytical path-based approach and the path-based simulation approach on the media multicast system.

CHAPTER 6

CONCLUSION AND FUTURE WORK

Solving large Markov models presents a fascinating challenge in managing storage complexity and achieving computational efficiency. In this dissertation, we present a novel path-based framework for analyzing a class of very large models. Through the notion of evaluating paths, our framework is the first to unify the traditionally disjoint solution techniques based on numerical analysis and simulation. Unlike previous path-based techniques, our framework is formulated at a higher level of abstraction, making it possible to construct approaches that can solve large models more efficiently.

In Chapter 2, we introduce the foundation for the framework and the basis for all path-based approaches presented in this dissertation. We show how a model can be analyzed at a higher level of abstraction than the tradition state-transition level. In particular, we implemented an approach for analyzing models at the component level. At that level of analysis, we show how the formulation of a partial order reduction relation among paths arises naturally. The approach is one of the first to use partial order reduction to compute Markov models efficiently. Through the use of the relation, multiple paths can be evaluated simultaneously.

Furthermore, many analytical techniques have difficulty solving large models, because they need to store both the transition matrices and the solution vectors. Even though there are a number of techniques for storing the matrices compactly, storing the the solution vectors still takes a large amount of memory for large models, and thus precluding many analytical techniques from being used. Our approach, however, computes the desired measures directly without ever storing those vectors or matrices. That makes it possible to solve very large models using a small amount of storage. As demonstrated in the chapter, the approach is able to solve a model that is several orders of magnitude larger than what could be solved by other techniques, and it uses only tens of megabytes of memory.

In Chapter 3, we present the path-composition algorithm. The algorithm is used to eliminate

redundant computation by computing a set of paths at a time from their subpaths. The subpaths are computed first, and then their results are used to compute the whole paths. Thus, by reusing computation, more paths more paths can be computed quickly. We also describe in this chapter an algorithm for computing efficiently a composable subpath class, wherein all subpaths are composable with each other. That class further helps the path-composition algorithm to work efficiently by computing only the necessary subpaths for composition to compute a set of whole paths. To the best of our knowledge, the work presented in this chapter is the first to propose the idea of path composition among work on path-based techniques. Overall the path-composition algorithm helps to improve performance and it achieves a speedup of 6.6 to 8.8 over the basic path-based approach described in 2.

Further performance improvement is necessary in order to make path-based approaches useful for analyzing large models. In Chapter 4, we discuss the idea of path selection. Since a path in our approaches is a high level abstraction of trajectories, there is more information available to use to select the important paths. The ability to select important paths helps in computing tight bounds quickly. We present three path-selection approaches and demonstrated their effectiveness in this chapter. The first two approaches prioritize the paths according to their reward and probability values. The third one dynamically switches between them based on how quickly the upper and lower bounds converges. We also compare the effectiveness of the approaches and discuss the classes of models that are most effectively evaluated by each approach. From our experiments, we have noticed that for some models, performance improves up to three orders of magnitude, bringing evaluation time from thousands of seconds down to a few seconds.

Although our analytical approaches can solve models that are already several orders of magnitude larger than what can be solved by other analytical techniques, there are many models that are simply too big to be solved efficiently by them. That is, we believe that our approaches can still be used to compute bounds for those model, but the bounds may be too wide to be useful. In Chapter 5, we present our path-based simulation approach for solving those very big models. Instead of computing bounds on the exact solution, the simulation approach computes confidence interval about the estimated mean of the solution. The main benefit of using the approach is that evaluating a path is equivalent to evaluating a set of trajectories by the traditional discrete-event simulation techniques. Thus, in addition to being more efficient, the approach is also more accurate, because it is based partially on numerical analysis. As shown

through our experimental results, it computes tight confidence intervals in a few seconds, whereas an analytical approach computes wide bounds and takes thousands of seconds.

In summary, we have presented a framework for solving large Markov models. We begin by presenting an path-based approach that can solve models that are much large than what can be solved by many other analytical techniques. Then, we present a number of algorithms for improving the performance of the approach. When the models to be solved are simply too large, we can resort to using our path-based simulation approach to compute confidence intervals about the true solution.

We intend to pursue future research agenda in the following directions:

- *Importance Sampling* For some models of rare events, the technique of importance sampling is used to estimate the probabilities of the rare events. The occurrences of the rare events are sped up by sampling a different distribution, and then the unbiased estimates of the events are recovered by multiplying by a likelihood ratio. Thus, the purpose of importance sampling is to reduce the variance in an estimator. Since our path-based framework is formulated at a higher level of abstraction, we believe there is more information available about the paths to help us construct better estimators for the rare events. The path-selection work presented in Chapter 4 is, in a sense, the precursor of the work on importance sampling. Importance sampling will help to improve the analysis of a large class of models.

- *Advanced Simulation* In Chapter 5, we present the path-based simulation approach wherein evaluating each path is equivalent to evaluating a set of trajectories by traditional discrete-event simulation techniques. The performance of that approach can be improved further by considering using the partial order reduction relation described in Chapter 2. That relation helps the simulation approach to sample *a set of paths simultaneously* when a single path is evaluated. That not only improves performance, but also allows more paths to be sampled in a given amount of time. In addition, by memorizing those paths that have already been sampled, the approach can reuse results and also determine to sample those paths that have not been sampled. Augmented with that capability, it can sample and cover a larger part of the path space.

- *Stationary Analysis* All of the approaches we have presented in this dissertation are used to compute transient measures. The other measure that is of interest, for example, in

availability and queuing models is stationary measure. In stationary analysis, we analyze the steady-state characteristics of a system. We are working toward augmenting the framework with the capability to solve models for steady-state measures.

REFERENCES

[1] V. V. Lam, P. Buchholz, and W. H. Sanders, "A structured path-based approach for computing transient rewards of large CTMCs," in *Proceedings of the First International Conference on the Quantitative Evaluation of Systems (QEST 2004)*, (Enschede, The Netherlands), IEEE Computer Society, September 2004, pp. 136–145.

[2] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley & Sons, Inc., 2002.

[3] L. Kleinrock, *Queueing Systems, Volume I: Theory*. John Wiley & Sons, Inc., 1975.

[4] W. J. Stewart, "A comparison of numerical techniques in Markov modeling," *Communications of the ACM*, vol. 21, pp. 144–152, February 1978.

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

[6] C. Moler and C. V. Loan, "Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later," *SIAM Review*, vol. 45, no. 1, pp. 3–49, 2003.

[7] J. G. Kemeny and J. L. Snell, *Finite Markov Chains*. D. Van Nostrand Company, 1960.

[8] P. Buchholz, "Exact and ordinary lumpability in finite Markov chains," *Journal of Applied Probability*, vol. 31, pp. 59–75, 1994.

[9] V. F. Nicola, "Lumping in Markov reward processes," IBM T. J. Watson Research Center, Yorktown Heights, NY, research report, 1989.

[10] S. Derisavi, H. Hermanns, and W. H. Sander, "Optimal state-space lumping in Markov chains," *Information Processing Letters*, vol. 87, no. 6, pp. 309–315, 2003.

[11] H. A. Simon and A. Ando, "Aggregation of variables in dynamic systems," *Econometrica*, vol. 29, pp. 111–138, 1961.

[12] P. J. Courtois, *Decomposability*. New York: Academic Press, 1977.

[13] H. H. Ammar and S. M. R. Islam, "Time scale decomposition of a class of generalized stochastic Petri net models," *IEEE Transactions on Software Engineering*, vol. 15, pp. 809–820, June 1989.

[14] W.-L. Cao and W. J. Stewart, "Iterative aggregation/disaggregation techniques for nearly-uncoupled Markov chains," *Journal of the ACM*, vol. 32, no. 3, pp. 702–719, 1985.

[15] P. J. Schweitzer, "Aggregation methods for large Markov chains," in *Mathematical Computer Performance and Reliability* (G. Iazeolla, P. J. Courtois, and A. Hordijk, Eds.), (Amsterdam, Holland), Elsevier, 1984, pp. 275–302.

[16] P. J. Schweitzer, "A survey of aggregation-disaggregation in large Markov chains," in *Numerical Solution of Markov Chains* (W. J. Stewart, Ed.), (New York), Marcel Dekker, Inc., 1991, pp. 373–384.

[17] R. Koury, D. McAllister, and W. J. Stewart, "Iterative methods for computing stationary distributions of nearly completely decomposable Markov chains," *SIAM Journal of Algebraic and Discrete Methods*, vol. 5, no. 2, pp. 164–186, 1984.

[18] A. Reibman, "A splitting technique for Markov chain transient solution," in *Numerical Solution of Markov Chains* (W. J. Stewart, Ed.), (New York), Marcel Dekker, Inc., 1991, pp. 373–384.

[19] P. Buchholz, "An adaptive decomposition approach for the analysis of stochastic Petri nets," *Performance Evaluation*, vol. 56, no. 1/4, pp. 23–52, 2004.

[20] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comp.*, vol. 35, pp. 677–691, Aug. 1986.

[21] A. Srinivasan, T. Kam, S. Malik, and R. E. Brayton, "Algorithms for discrete function manipulation," in *Proc. of the Int'l Conf. on CAD (ICCAD'90)*, 1990, pp. 92–95.

[22] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Information and Computation*, vol. 98, pp. 142–170, June 1992.

[23] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[24] G. Ciardo and A. S. Miner, "A data structure for the efficient Kronecker solution of GSPNs," in *Proc. 8th Int. Workshop Petri Nets and Performance Models*, 1999, pp. 22–31.

[25] S. Derisavi, P. Kemper, and W. H. Sanders, "Symbolic state-space exploration and numerical analysis of state-sharing composed models," *Linear Algebra and Its Applications (LAA)*, vol. 386, pp. 137–166, July 2004.

[26] B. Plateau, "On the stochastic structure of parallelism and synchronization models for distributed algorithms," *Performance Evaluation Review*, vol. 13, no. 2, pp. 147–154, 1985.

[27] P. Buchholz, "An aggregation/disaggregation algorithm for stochastic automata networks," *Probability in the Engineering and Informational Sciences*, vol. 11, pp. 229–253, 1997.

[28] P. Buchholz, "Structured analysis approaches for large Markov chains," *Applied Numerical Mathematics*, vol. 31, pp. 375–404, 1999.

[29] W. J. Stewart, K. Atif, and B. Plateau, "The numerical solution of stochastic automata networks," *European Journal of Operation Research*, vol. 86, pp. 503–525, 1995.

[30] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper, "Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models," *INFORMS Journal of Computing*, vol. 13, no. 3, pp. 203–222, 2000.

[31] G. Ciardo and M. Tilgner, "On the use of Kronecker operators for the solution of generalized stochastic Petri nets," NASA Langley Research Center, Hampton, VA, Tech. Rep. ICASE Report, 1996.

[32] P. Kemper, "Numerical analysis of superposed GSPNs," *IEEE Transactions on Software Engineering*, vol. 22, pp. 615–628, September 1996.

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

[34] E. de Souza e Silva and P. M. Ochoa, "State space exploration in markov models," in *ACM SIGMETRICS '1992 International Conference on Measurements and Modeling of Computer Systems*, vol. 20 of *Performance Evaluation Review*, June 1992, pp. 152–166.

[35] J. L. Bon and J. Collet, "An algorithm in order to implement reliability exponential approximations," *Reliability Engineering and Safety Problems*, vol. 43, no. 3, 1994.

[36] D. M. Nicol and D. L. Palumbo, "Reliability analysis of complex models using SURE bounds," *IEEE Transactions on Reliability*, vol. 44, pp. 46–53, March 1995.

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

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

[39] E. de Souza e Silva and H. R. Gail, "Calculating transient distributions of cumulative reward," in *Proceeding of the ACM SIGMETRICS 1995 International Conference on Measurements and Modeling of Computer Systems*, May 1995, pp. 231–240.

[40] M. A. Qureshi, "Construction and solution of Markov reward models,", Ph.D. dissertation, University of Arizona, 1996.

[41] M. A. Qureshi and W. H. Sanders, "A new methodology for calculating distributions of reward accumulated during a finite interval," in *Proceedings of the 26th International Symposium on Fault-Tolerant Computing*, 1996, pp. 116–125.

[42] A. P. A. van Moorsel and B. R. Haverkort, "Probabilistic evaluation for the analytical solution of large Markov models: Algorithms and tool support," *Microelectronics and Reliability*, vol. 36, no. 6, pp. 733–755, 1996.

[43] S. Derisavi, P. Kemper, and W. H. Sanders, "Symbolic state-space exploration and numerical analysis of state-sharing composed models," in *Proceedings of NSMC '03: The Fourth International Conference on the Numerical Solution of Markov Chains*, 2003, pp. 167–189.

[44] W. Grassmann, "Means and variances of time averages in Markovian environments," *European Journal of Operational Research*, vol. 31, no. 1, pp. 132–139, 1987.

[45] F. Bause, P. Buchholz, and P. Kemper, "A toolbox for functional and quantitative analysis of DEDS," in *Quantitative Evaluation of Computing and Communication Systems*, Springer, 1998, LNCS 1469, pp. 356–359.

[46] P. Buchholz and P. Kemper, "Numerical analysis techniques in the APNN toolbox," in *Proceedings of the Workshop on Formal Methods in Performance Evaluation and Applications*, 1999, pp. 1–6.

[47] D. Daly, D. D. Deavours, J. M. Doyle, P. G. Webster, and W. H. Sanders, "Möbius: An extensible tool for performance and dependability modeling," in *Computer Performance Evaluation: Modelling Techniques and Tools*, Springer, 2000, LNCS 1786, pp. 332–336.

[48] R. R. Muntz and J. Lui, "Computing bounds on steady-state availability of repairable computer systems," *Journal of the ACM*, vol. 41, no. 4, pp. 676–707, 1994.

[49] A. P. A. van Moorsel and W. H. Sanders, "Adaptive uniformization," *ORSA Communications in Statistics: Stochastic Models*, vol. 10, no. 3, pp. 619–648, 1994.

[50] P. Kemper, "Transient analysis of superposed GSPNs," *IEEE Transactions on Software Engineering*, vol. 25, no. 2, pp. 182–193, 1999.

[51] A. S. Miner, "Computing response time distributions using stochastic Petri nets and matrix diagrams," in *Proc. 13th Int. Workshop on Petri Nets and Performance Models (PNPM'03)*, 2003, pp. 10–19.

[52] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. MIT Press, 1990.

[53] Y. Chu, S. G. Rao, and H. Zhang, "A case for end system multicast," in *Proceedings of the ACM SIGMETRICS 2000 International Conference on Measurements and Modeling of Computer Systems*, 2000, pp. 1–12.

[54] R. A. Howard, *Dynamic Probabilistic Systems, Vol. I, II*. John Wiley & Sons, 1971.

[55] B. L. Fox and P. W. Glynn, "Computing Poisson probabilities," *Communications of the ACM*, vol. 31, pp. 440–445, April 1988.