AN ABSTRACT OF THE DISSERTATION OF

Scott Alan Burgess for the degree of Doctor of Philosophy in Computer Science presented on August 31, 2001. Title: Analysis of Bayesian Anytime Inference Algorithms.

Abstract approved: Redacted for Privacy

Bruce D. D'Ambrosio

This dissertation explores and analyzes the performance of several Bayesian anytime inference algorithms for dynamic influence diagrams. These algorithms are compared on the On-Line Maintenance Agent testbed, a software artifact permitting comparison of dynamic reasoning algorithms used by an agent on a variety of simulated maintenance and monitoring tasks. Analysis of their performance suggests that a particular algorithmic property, which I term *sampling kurtosis*, may be responsible for successful reasoning in the tested half-adder domain. A new algorithm is devised and evaluated which permits testing of sampling kurtosis, revealing that it may not be the most significant algorithm property but suggesting new lines of inquiry. Peculiarities in the observed data lead to a detailed analysis of agent-simulator interaction, resulting in an equation model and a Stochastic Automata Network model for a random action algorithm. The model analyses are extended to show that some of the anytime reasoning algorithms perform remarkably near optimally. The research suggests improvements for the design and development of reasoning testbeds.

Copyright by Scott Alan Burgess
August 31, 2001
All Rights Reserved

Analysis of Bayesian Anytime Inference Algorithms

by

Scott Alan Burgess

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented August 31, 2001
Commencement June 2002

Doctor of Philosophy dissertation of Scott Alan Burgess presented on August 31, 2001

APPROVED:

Redacted for Privacy

Major Professor, representing Computer Science

Redacted for Privacy

Head of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Redacted for Privacy

Scott Alan Burgess, Author

ACKNOWLEDGEMENTS

If you are reading this and you are not a member of the committee that will determine whether I receive the PhD degree then rest assured, dear soul, you are beginning in the right place. Much of this book is written for *you*, a student or researcher who wishes to learn a little more about what I labored on for so many years. While you will probably be one of only a few to ever read it, please use my bibliography and text as best you can. You won't be standing on the shoulders of a giant, but a three-inch-thick book is better than nothing. Also, I would commend to you a wonderful text [Tufte 83] that may help you design all of the data graphs you will create for your own work. See, there is something worth reading in the acknowledgements!

I would like to thank Dr. Bruce D'Ambrosio for his guidance and insight, and for being a bottomless well of patience; Dr. Paul Cull for instruction that has extended far beyond the classroom; Drs. Michael Quinn and Prasad Tadepalli for their input, accessibility, and help in innumerable ways; Dr. Margaret Niess, my original graduate committee representative, for her professionalism and gentle manners; and Dr. Albert Stetz who graciously agreed to fill in as my graduate committee representative on short notice. Thanks too to Dr. Billy Stewart, who wrote an excellent text on Markov chains, and who was kind enough to discuss his research with me when I called him out of the blue.

I would also like to thank the students who have journeyed here before me and with me: your friendship, help, and companionship will be missed when I leave.

This dissertation is dedicated to the memory of my father, Major Roy Walter Burgess, who wished dearly that he could be here to see me graduate, and to those of us

who survive him: my brothers Mike, Steve, and Mark; and my loving mother, Betty Burgess, who won't understand a word of this silly book but will love it and me just the same.

TABLE OF CONTENTS

Page

1. Introduction....1

1.1 Motivation....1

1.2 Significance....2

1.3 Contributions of the Dissertation....3

1.4 Organization of the Dissertation....5

2. Previous Work....7

2.1 Introduction....7

2.1.1 Fundamental Definitions....7
2.1.2 Bayesian Networks....9
2.1.3 Influence Diagrams....10
2.1.4 Cooper's Transformation....12
2.1.5 Dynamic Influence Diagrams—Inference Over Time....13

2.2 Algorithms for Resource-Bounded Inference....15

2.2.1 Backward Simulation....15
2.2.2 The Kappa-Reduced Algorithm....16
2.2.3 Posterior Kappa-Reduced....17
2.2.4 D-IPI....19
2.2.5 Random....19

2.3 Partially Observable Markov Decision Processes....20

2.4 Stochastic Automata Networks....23

3. The On-Line Maintenance Agent....28

3.1 Imposing Structure on Real-Time Inference....28

3.2 The Task Level....28

3.3 The Agent-Simulator Level....29

3.4 The Component Level....32

TABLE OF CONTENTS (Continued)

Page

3.5 The Action Level..33

4. Explorations With Several Inference Algorithms....................................36

4.1 Experimental Goals..36

4.2 Method...37

4.3 Results...42

4.4 Discussion..47

4.5 Related Results...49

4.6 Conclusions..50

5. Some Experiments On Kurtosis..52

5.1 Experimental Goals..52

5.2 Method...52

5.3 Results...56

6. Analysis of the Random Algorithm..63

6.1 Introduction..63

6.2 Derivation of the Basic Cost Equations for the Random Algorithm.....64

6.3 Analysis of the Cost Equations for the Random Algorithm...............68

6.4 A Stochastic Automata Network Model..73

6.5 Models for Agent and Simulator Speed Ratios.................................76

6.6 Transition Matrices for Submodels of the Stochastic Automata Network 80

6.7 Evaluation of the SAN Model...82

6.8 Conclusions..86

TABLE OF CONTENTS (Continued)

Page

7. Conclusions 88

    7.1 Summary of Findings 88

    7.2 Future Research 89

Bibliography 91

APPENDICES 96

LIST OF FIGURES

Figure		Page
2.1	A Simple Bayesian Network	10
2.2	An Influence Diagram With a Single Decision D And a Value Node V	12
2.3	A Dynamic Influence Diagram Showing Two Decision Cycles	14
3.1	The Simulator's Half-adder Circuit	33
3.2	Abstraction Basis For an Agent Decision	34
4.1	Cost Per Unit Failure of the Kappa-reduced Algorithm Across Step Size	43
4.2	Cost Per Unit Failure of the D-IPI Algorithm Across Step Size	44
4.3	Cost Per Unit Failure of the Posterior Kappa-reduced Algorithm Across Step Size	45
4.4	Comparison of the Best Average Performance of Several Anytime Algorithms and Exact Inference	46
5.1	Average Performances of Backward Simulation with 1000 Samples at Different Kurtosis Values	56
5.2	Critical Region of the Averaged Performance Data for the Backward Simulation Algorithm with 1000 Samples at Different Kurtosis Values	57
5.3	Average Performance of Backward Simulation with 5000 samples at Different Kurtosis Values	59
5.4	Average Performance of Backward Simulation with 20000 Samples at Different Kurtosis Values	60
6.1	Comparison of the Random Action Algorithm with Sample Size 5000 Data with Theoretical Predictions	69
6.2	Cost Per Unit Failure of the Random Algorithm at 1000 Samples with 90% Confidence Intervals and Variances	71
6.3	Graphical Representation of the Decision Process When Agent and Simulator Run at the Same Speed Using the Random Algorithm	77

LIST OF FIGURES (Continued)

Figure | Page

6.4 Graphical Representation of the Decision Process When Simulator Speed is Twice Agent Speed, With the Agent Using the Random Algorithm 78

6.5 Generalizing of the Decision Process Model for Simulator/Agent Speed Ratios Greater than 1:1 79

6.6 Comparisons Between Stochastic Automata Network Predictions of Random Performance and Actual Random Performance 83

6.7 Comparison of Two Perfect Agent Models With Performance of Several Anytime Inference Models 85

LIST OF TABLES

Table — Page

4.1 Comparison of Best Average Performance of Several Anytime Algorithms and Exact Interface....46

5.1 An Example of Peaking a Distribution with Logic Sampling....53

LIST OF APPENDICES

Page

Appendix A: Cost-Per-Unit-Failure Performance Data for Several Anytime Inference Algorithms.....97

A.1 Introduction.....97

A.2 Data for the Kappa-reduced Algorithm.....97

A.3 Data for the D-IPI Algorithm.....98

A.4 Data for the Posterior Kappa-reduced Algorithm.....99

A.5 Data for the Exact Algorithm.....100

Appendix B: Raw Data for Experiments with the Peaked Backward Simulation Algorithm.....102

B.1 Introduction.....102

B.2 The Raw Data.....103

Appendix C: Results of the Random Action Algorithm for Samples of Sizes 1000 and 5000.....124

C.1 Introduction.....124

C.2 Runs With 1000 Samples.....125

C.3 Runs With 5000 Samples.....135

C.4 Calculation of Means and Confidence Intervals for Samples of Size 1000.....137

Appendix D: Code for Calculations Involving One Submodel of the Complete Stochastic Automata Model for the OLMA Running the Random Algorithm.....138

D.1 Matlab Code for Calculating the Stochastic Automata Network.....138

LIST OF APPENDIX TABLES

Table Page

A.1 Averages of Performance Data for the Kappa-reduced Algorithm 97

A.2 D'Ambrosio's Averages of Performance Data for the Kappa-reduced Algorithm as Published in UAI-96..98

A.3 Performance Data for the D-IPI Algorithm...98

A.4 D'Ambrosio's Performance Data for the D-IPI Algorithm as Published in UAI-96..99

A.5 Average Performance Data for the Posterior Kappa-reduced Algorithm...........99

A.6 D'Ambrosio's Average Performance Data for the Posterior Kappa-reduced Algorithm as published in UAI-96..100

A.7 Average Performance Data for the SPI Exact Algorithm.........................101

A.8 Minimum Cost-Per-Unit-Failure Values for the SPI Exact Algorithm as Published in UAI-96...101

C.1 Calculation of Means and Confidence Intervals for Samples of Size 1000.......137

Ὡς ἡδύ τόν σωθέντα μεμνῆσθαι πόνου —*Euripides.*

Analysis of Bayesian Anytime Inference Algorithms

1. Introduction

1.1. Motivation

This dissertation empirically explores and evaluates performance of Bayesian anytime inference algorithms. Bayesian networks have been shown to be a robust model of uncertainty and causal influence for many machine-reasoning tasks. Once generated and instantiated, many inference algorithms typically can produce desired results. However, little is known about *which* algorithm to use in a particular context.

The question is important because of the general intractability of Bayesian inference [Cooper 90]. While Bayesians may effectively model domains, generating answers in large domains may be more difficult. Anytime algorithms partially answer this question by incrementally building an answer. But while a partial answer is available at any time, the user must question whether a partial answer is useful.

I initially began looking at the problem with the idea that perhaps additional speedup could be gained through parallelizing some of these algorithms, but abandoned this approach since linear speedup promises little return when computation grows exponentially with the size of the problem, and since generating such speedup would not address issues of the quality of decisions.

I gradually realized there was also no obvious way to rigorously compare existing algorithms in order to analyze their behavior. There were simply too many variables: what the agent was reasoning about, how time and objects were modeled, which

inference algorithms were used, how the processing was divided between the agent and the simulation, tuning algorithms to perform their best, tuning different algorithms to the "same" degree, and so forth. The quantity of data required to adequately cover so many variables was completely unmanageable.

I reasoned that perhaps I could test whether certain properties were important to the performance of these algorithms. By eliminating some possibilities and confirming others, I could narrow the quest for better algorithms to more profitable areas. This could ultimately result in guidelines that would loosely govern the design of approximation algorithms for inference.

1.2. Significance

Critics of the history of artificial intelligence reasonably point out that most problems solved so far have been in overly simplified environments, or "blocks worlds." It is more widely understood today that a critical issue for artificial intelligence is scaling up solutions to tackle problems with real-world scale [Schank 91]. Attempts to move reasoning out of the lab and into the world must face the enormous complexities the world contains. Deliberative agents that worked in a "blocks-world" suffer from two unscalabilities: the world they operated in inadequately represented the complexities of the real world, and the real world demands action rather than unbounded deliberation. A robot ordered to fetch a cup in an adjacent room may go there only to find that the cup does not rest on the expected table and looks like a stein. A web engine searching for patriarchs of the Eastern Orthodox Church will have to contend with human differences of opinion about which lineage represents the true patriarchs, yet still answer quickly.

Note that the problem is not restricted to creating a set of static policies. An enormous amount of real world reasoning requires first formulating an appropriate model of the problem, usually in a short space of time.

I therefore believe it is beneficial to view reasoning as a dynamic activity involving model formulation and decision-making under uncertainty. To that end, my research attempts to make reasoning more tractable, even when faced with tight time constraints, decisions, utilities, uncertainties, and dynamically generated models.

Thus this dissertation addresses issues of scale in dynamic reasoning that must be faced in order for artificial intelligence to advance as a field.

1.3. Contributions of the Dissertation

I have generated several contributions comprising two primary experiments and two models of performance. The first experiments of the dissertation compared several incremental inference algorithms. The results are sufficient to confirm that most incremental algorithms are effective at trading time for quality of inference. Several algorithms worked well and one remarkably well, despite an extremely small allowance of time to perform inference. A follow up experiment revealed that the best performer in the primary experiments was not so in all domains. Some of the algorithms required a significant amount of tuning to achieve optimal performance for a particular CPU speed. Consistency of performance was more problematic than expected for some algorithms, but the best algorithm also proved to be a remarkably consistent performer.

The second set of experiments analyzed one hypothesized critical component of performance. Since the first set of experiments suggested that focusing on probability

mass was critical for making good decisions, I designed an algorithm that had a tunable focusing mechanism. The hope was that this would demonstrate the importance of this property in incremental inference, but the improvement with tuning is insufficiently dramatic to support any broad conclusions. Focusing does appear to improve performance, but less than twenty percent. And stochastic simulation appears to be ineffective as an approximation algorithm for real-time inference. This does have implications for the design of effective anytime inference algorithms, as this algorithm scales well to very large networks. It is not known how well the other algorithms can scale to large networks for use in common domains.

More important than the results of either set of experiments is the evolution of experimental methodology. I pressed the early experiments to acquire accurate data, but found later that the methodology had not accounted for how I planned to use the data later. I also saw an important flaw in the testbed: updating the network for each time cycle is done using SPI exact inference, regardless of the algorithm used for decision making. While this is justifiable using the current circuit simulation in the testbed, this approach may not scale well as the networks become larger and more highly connected.

While I could not afford to rework the testbed to address the updating flaw, data quality was addressable through better experimental design. I chose to collect sufficient quality and quantity of data to permit a basic statistical comparison of the performance of different algorithms. This required researching the statistical literature for nonparametric methods for comparing the means of two data sets when the number of samples of each data set was small—hence my choice of the Wilcoxen rank-sum test. From earlier experiments I surmised correctly that averaging four runs for each setting of parameters

would be sufficient for good statistical analysis of the results. Since I still could not compare different algorithms for the reasons already described, designing an individual algorithm to test a particular property eliminated many possible variables.

Despite the mediocre performance of the algorithm I designed, I believe the results substantiate the effectiveness of the methodology. The results also demonstrate the flexibility of a testbed that was not originally intended for the purposes I put it to.

To answer certain questions concerning optimal performance, I decided to attempt modeling the testbed when a random decision algorithm was used. While this initially appeared straightforward, I found that a behavioral equation model was inadequate for predicting performance, and clumsy for many calculations. My investigations in the literature uncovered the stochastic automata network model, which generated better prediction of performance and better scalability for calculations. Together the two models provide an adequate summary of the testbed's behavior under the random algorithm and under ideal conditions.

I found that the testbed itself could be improved for supporting research in inference algorithm comparison. In particular, the current model of time in the testbed could be improved. And my research suggests that a testbed should be designed in conjunction with a theoretical model so that its performance may be more easily evaluated.

1.4. Organization of the Dissertation

The rest of this dissertation is organized as follows. The early chapters present relevant background materials. Chapter 2 introduces some basic elements of Bayesian networks,

influence diagrams, and related work on the inference algorithms and advanced structures used to conduct the experiments. It also briefly discusses partially observable Markov decision problems and stochastic automata networks (SANs). Chapter 3 introduces the On Line Maintenance Agent testbed I used to conduct my research.

Chapters 4 through 6 present the main research results. Chapter 4 presents the early experiments on the tradeoffs between time to solution and quality of solution with incremental algorithms. The discussion states the general problem, then the experimental design, and finally the results and conclusions drawn from the experiment. Following the same structure as the previous chapter, Chapter 5 presents the experiment that tests the value of a focusing mechanism in the approximation algorithms. Chapter 6 discusses a theoretical model of performance utilizing equations derived from the structure of the On-Line Maintenance Agent, and another utilizing the stochastic automata network model.

Chapter 7 presents the final conclusions and a discussion of future work in this area.

2. Previous Work

2.1. Introduction

2.1.1. Fundamental Definitions

While my central aims are representing beliefs in various propositions over time and using these beliefs to make intelligent decisions, there are numerous methods of representing uncertainty. There is simply not room enough here to outline the other methods and compare their relative merits. Nor do I intend here to introduce all details of the Bayesian framework. This chapter does provide the notation that I intend to use for the remainder of this dissertation, and it provides details of some algorithms and past work less commonly discussed.

For introductory articles, see [Charniak 91] or [Heckerman 95]. More thorough presentations are made in [Pearl 88], [Neapolitan 90], [Russell et al. 95] and [Jensen 96]. [D'Ambrosio 99] provides a more recent view of Bayesian reasoning with greater emphasis on some of the algorithms discussed in this dissertation.

Let us consider an air traffic control equipment scenario I became familiar with while working for the Federal Aviation Administration. Radar, communication and other field equipment not only reports information to a central control center for use in directing aircraft, but it also is maintained and repaired from a (possibly separate) central repair facility. The central repair facility receives constant updates concerning the state of equipment. When fault information arrives, the person receiving it has several choices: ignore the broken equipment for now if it is nonessential, attempt to do partial

diagnosis of the problem (providing the equipment has this capability), switch to a parallel piece of equipment which can perform the same function (providing that parallel equipment is available in the field), or send out a repair crew.

The choices may be difficult. There are many different types of equipment in the field, and even an experienced central controller may not know the diagnostic routines for a particular piece of equipment in the field. But sending a repair crew can be quite costly, since equipment may be hundreds of miles from the facility. The costs may be compounded if the crew does not have the proper tools to effect repairs on their first visit to the equipment. And all of this must be weighed against the current state of neighboring equipment and the safety of air traffic in the region.

From this we can see the dynamic nature of diagnostic reasoning. The central repair controller may probe and evaluate the equipment a little before selecting a course of action. His decision will be made in the context of information obtained and may change if further information changes his beliefs about the equipment or its priority.

In a probabilistic representation, we might create a belief state variable F with values YES and NO depending on whether the repair controller believes there is a failure, and an observation variable O with states RED LIGHT and OKAY which represents the repair controller's console observations. The controller would like to know how likely a fault is given his observations. We can represent the updated belief in the probability of failure $P(F)$ according to the context O by writing $P(F|O)$ and interpreting this to mean the probability of observing failure F given the particular failure context O.

In order to define the value of $P(F|O)$, we may use the equation of Reverend Bayes:

$$P(F \mid O) = P(F,O)/P(O)$$ **Equation 2.1**

We can think of F as an event prior to O, which influences the likelihood we will observe the states of O. The revised probability is the result of dividing the *joint probability* $P(F,O)$ by the probability of different observations.

2.1.2. Bayesian Networks

While the example above is sufficient when dealing with a very small number of variables, one would like a representation more open to examination and revision when the number of variables grows. We can represent probabilistic information graphically by using a *Bayesian network*. A Bayesian network is defined as an acyclic graph with:

- A set of *chance* nodes, each node representing a random variable comprising a finite number of mutually exclusive states
- A set of arcs, each arc between a single pair of nodes, with the intuitive meaning that the random variable at the arc's source directly influences the variable at its tip, and
- A set of conditional probability distributions, one for each node, where the probability distribution for a node quantifies the effect of the graphical predecessors on the states of that node

An example for the scenario discussed in Section 2.1.1 appears in Figure 2.1. Notice that the evidence observed by the controller is directly influenced by whether a fault has occurred. Often Bayesians will show the graphical structure of a problem without showing the individual probabilities, since the graphical structure encodes the causal independences between variables and since there are often a lot of numbers!

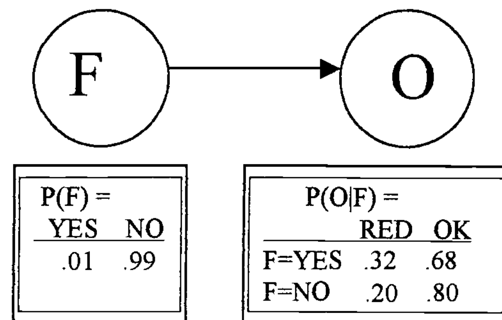

Figure 2.1: A Simple Bayesian Network.

2.1.3. Influence Diagrams

While the example above allows the controller to model his beliefs about the states of variables, it is insufficient for allowing a machine to make decisions based upon the model. To do this, we need to add decisions and utilities to our graphical model.

Decisions are modeled graphically by adding a square for each decision that can be made. Observations that precede a decision (and are therefore available to the decision-maker) are connected to the decision node by arcs leading to it. Decisions affecting the outcome of some states are connected by arcs leading from the decision node to those states.

Utilities are an encoding of our expert's believed costs for fault states and various actions. Taking a repair action may have a high cost. If a part is not likely failing, the controller would probably not replace it. But system outages in an air traffic control scenario may have rare but extremely high costs. How does our controller decide what to do? What he needs to know to make a good decision is:

- How likely each resulting situation is, given the observations he has made and the actions he can take
- The cost or utility of each of these possible resulting situations

Given the likelihood of a particular outcome and its utility, the expected utility is simply the product of both.

Let $\hat{S} = \{S_i\}$ be the set of possible resulting states from the set $\hat{O} = \{O_m\}$ of possible observations and the set $\check{D} = \{D_k\}$ of decisions. Let $\hat{U} = \{U_i(S_i)\}$ be the set of utility values associated with each resulting state. Then the best decision from a rational point of view is the one that maximizes expected utility. This is called the Maximum Subjective Expected Utility. We may write this as:

$$MSEU = \underset{i}{MAX}\left(\sum_{j} P\left(S_j \mid \hat{O}, D_i(S_j)\right) \bullet U_i(S_j)\right) \quad \textbf{Equation 2.2}$$

The graphical counterpart to Equation 2.2 is illustrated in Figure 2.2. Here the rectangular *decision node* represents the maximization necessary to pick the best decision. The diamond-shaped *value node* represents the utility associated with a particular state of some variables and/or decisions. The state of variable S_1 provides information to the decision node D. Both variable S_1 and D influence the state of variable S_2. The decision made and the final state of S_2 influence the utility. If S_1 and S_2 are components in the air traffic control system, then the first component may adversely affect the state of the second, and the controller must make a decision about replacing the second component in order to minimize the quantified costs and threats.

Multiple decision nodes and multiple value nodes may be incorporated to handle more complex decisions.

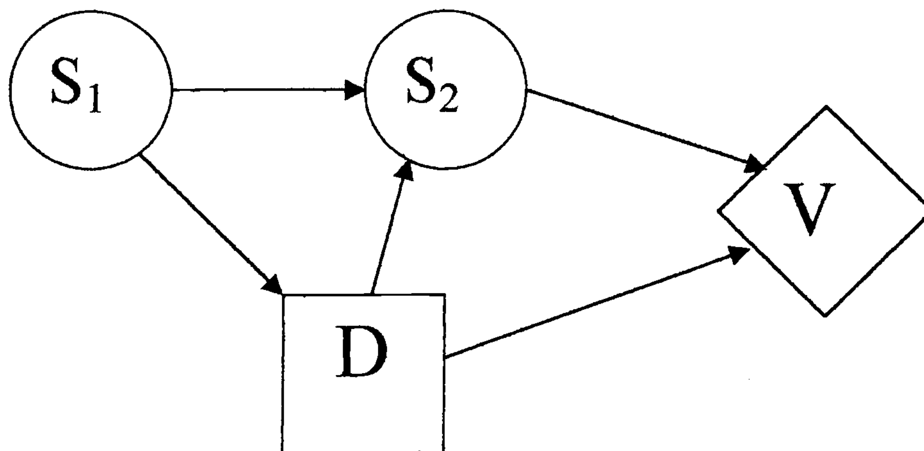

Figure 2.2: An Influence Diagram With a Single Decision D and a Value Node V.

2.1.4. Cooper's Transformation

While many algorithms exist for calculating specific probability queries in Bayesian networks, algorithms for answering decision queries in influence diagrams are more complicated and often utilize Bayesian inference algorithms as components. To simplify this situation, [Cooper 89] presents a transformation from influence diagrams to Bayesian networks that permits use of Bayesian network algorithms for decision problems.

To effect this transformation, we alter the influence diagram so that both value nodes and decision nodes are transformed into chance nodes. To transform a value node into a chance node we linearly map the utility values into a probability function and change the representation into a chance node in the graph. The chance node representing the value node will be a binary node with states T and F, and the probability of T is defined to be $P(V=T|v(P(v))) = [v(P(v)) + k_2] / k_1$ where $v(P(v))$ is the value function

indexed by the parents of V, k_1 is the difference between the maximum and minimum values $v(P(v))$ returns, and k_2 is the minimum value $v(P(v))$ returns.

To transform a decision node into a chance node, we need to assign a prior probability to each decision (the exact value generally doesn't matter as long as the sum of values is 1), and change its graphical representation.

Now we may solve the decision problem by instantiating each decision node to each possible value and solving for the maximum expected value. The decision instantiation that produces the maximum expected value for the evaluation of the belief network is the same as the decision we would make in the influence diagram representation using an influence diagram algorithm.

This transformation does have its limitations, however. While it is used extensively in this dissertation, I have found a modification that simplifies evaluation of decisions for the backward simulation algorithm: instead of assigning arbitrary priors to decision nodes, assign a uniform distribution to the possible decisions. Now a simulation algorithm need not evaluate one Bayesian network for each possible decision since randomly selecting a decision as a true chance node will converge to the same outcome.

2.1.5. Dynamic Influence Diagrams—Inference Over Time

In some cases, a single decision may be insufficient for solving a problem. Consider the case where our air traffic repair controller must make multiple repairs on the same piece of equipment before it can be fully restored. Normally such compound repair actions are not modeled in influence diagrams, as the growth in the number of possible actions adversely affects computing a solution and such multiple repair actions may be

unrealistic in many domains. Thus a common solution is to model only single repair actions over multiple decision cycles. To do this, we must again modify our original representation.

Figure 3 presents a general *dynamic influence diagram* for decision making. The states of multiple components are represented by the collective state nodes S_i that may produce observations O_i available to the decision-making agent. Generally the agent would like to make an optimal or approximately optimal first decision, since evidence available at future states cannot necessarily be predicted in a domain where components can actually break.

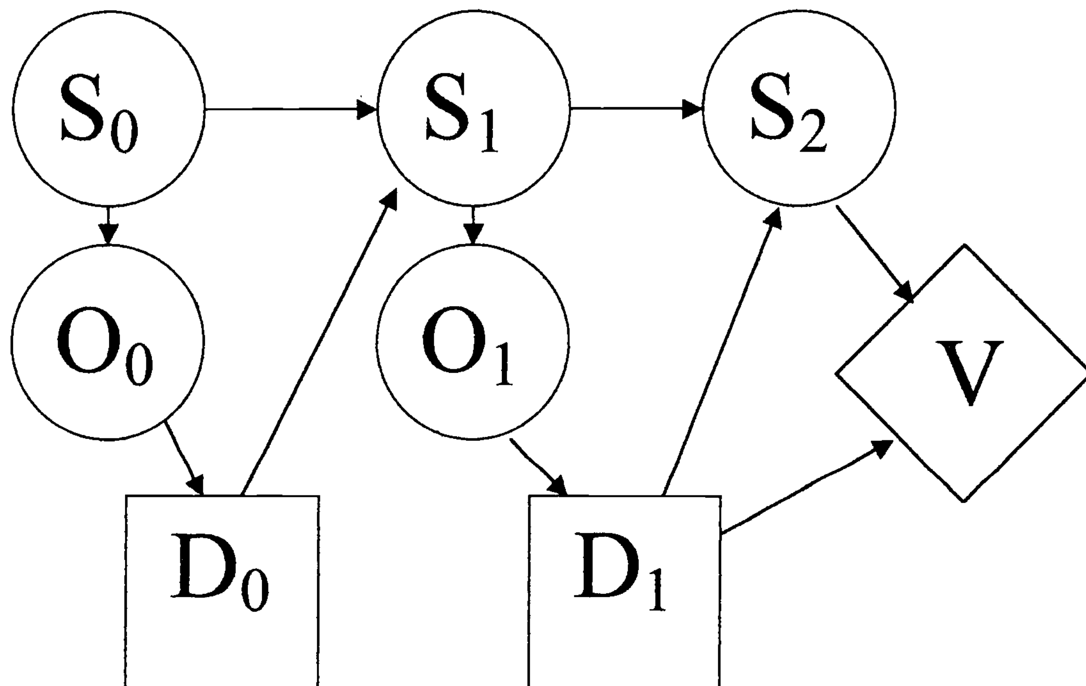

Figure 2.3: A Dynamic Influence Diagram Showing Two Decision Cycles.

To do inference over time, a best possible decision is made during the current time slice available. Next, the network is extended one time slice and updated to reflect the effects of the decision made. In this dissertation we also perform a folding in of the last time step into a small number of predecessor nodes that reflect the past states. Where time and storage space are not issues, the entire time slice may be stored away for future examination.

The mechanics of these operations can be quite inelegant and can vary in implementation, so we skip the details here. There is no reason to suppose that my results should be affected by implementation differences.

2.2. Algorithms for Resource-Bounded Inference

Numerous algorithms that attempt to trade accuracy for time have been developed in the last dozen years. While this section is not an exhaustive list of those, it does introduce the algorithms discussed in the dissertation, plus a few others that represent common or significant approaches to the problem.

2.2.1. Backward Simulation

Stochastic simulation algorithms are used in a variety of domains to estimate probabilities. They may likewise be used to estimate the MSEU in an influence diagram or a Bayesian network. To create a single instantiation of a Bayesian network, the root nodes are first sampled, then the children of those nodes, and so on until a complete instantiation of the network is created.

This can be remarkably inefficient when dealing with evidence. Recall from our simple network in Figure 1.1 and the ensuing discussion that evidence nodes are usually children of the state nodes. When a network is structured this way, samples inconsistent with the evidence are thrown away. The percentage of samples discarded is often large when evidence suggests a diagnosis that has low prior probability.

To avoid this we may use *backward simulation* [Fung 94]. Backward simulation performs a stochastic simulation by working backwards from evidence nodes to root nodes, then forward to the leaves. An evidence node is sampled using its conditional distribution in order to instantiate its parents. The value saved is then weighted by unsampled parent node prior distributions (normally a subset of the root nodes). This guarantees convergence to the true sample value without wasting samples.

2.2.2. The Kappa-Reduced Algorithm

Bruce D'Ambrosio implemented this algorithm upon suggestions from Moises Goldszmidt that are based upon the Kappa calculus [Goldszmidt 95]. Utilizing the partial ordering of the belief network that is already maintained, computation proceeds as follows:

1. For each node in order, compute its priors:

   $$P(n_i) = P(n_i \mid parents(n_i)) \prod_{p \in parents(n_i)} P(n_p)$$

2. Find the highest probability value in each node, and then find the smallest value in this set (the *least-greatest-prior*).

3. Build an ordered list of all of the probabilities that were computed in the first step, eliminating duplicates.
4. Set the *current-minimum-prior* to be the least-greatest-prior for the first iteration, or the next smaller entry in the sorted list for each subsequent iteration.
5. Beginning with the complete conditional probability table at each node, reduce the domain at each node to include only values greater than or equal to the current-minimum-prior.
6. Apply exact inference to the resulting reduced network to obtain a decision. Our version uses SPI exact inference [Li and D'Ambrosio 94].
7. If you desire another iteration, GO TO STEP 4.

The anytime capability of the algorithm stems from the ability to vary the number of loops on items four through seven. Such a loop is referred to later as a *step*. This algorithm exhibited weak performance in practice, so a posterior-estimating version was created as follows in the next section.

2.2.3. Posterior Kappa-Reduced

This variant makes only a few changes to the algorithm above. Computation proceeds as follows:

1. Reduce evidence nodes and their parents by eliminating all values inconsistent with current evidence.

2. For each node in order, compute its priors:

$$P(n_i) = P(n_i \mid parents(n_i)) \prod_{p \in parents(n_i)} P(n_p)$$

3. For each node prior to the evidence nodes, in reverse order from the evidence nodes, compute: $P(parent(n_i))\prod_{n_i} P(n_i \mid parent(n_i))$.

4. Find the highest of these values at each node, and then find the smallest value in this set (the *least-greatest-posterior*).

5. Sort all posteriors from all nodes into descending order in a list, eliminating duplicates.

6. Set the *current-minimum-posterior* to be the least-greatest-posterior for the first iteration, or the next smaller entry in the sorted list for each subsequent iteration.

7. Beginning with the complete conditional probability table at each node, reduce the domain at each node to include only values greater than or equal to the current-minimum-posterior.

8. Apply exact inference to the resulting reduced network to obtain a decision. Our version uses SPI exact inference [Li and D'Ambrosio 94].

9. If you desire another iteration, GO TO STEP 6.

The passing of evidential influence is analogous to lambda message propagation in Pearl's polytree algorithm [Pearl 88], and D'Ambrosio and I did consider implementing that as an alternative, but we concluded that it should make no significant difference in either decisions or performance of the algorithm, and used the version

described above. As with the Kappa algorithm, a *step* refers to an iteration of the loop from items six through nine above.

2.2.4. D-IPI

This algorithm extends the IPI algorithm presented in [D'Ambrosio 93] to cover search over more general expressions that include sum, difference, and maximization operators. IPI is an incremental search variant of the SPI exact algorithm mentioned above. Computation proceeds as follows:

1. Construct a symbolic expression for the query (marginalization over the joint probability density function).
2. Construct an evaluation tree for the query.
3. Search the tree top-down for large-valued joint instantiations of the variables.
4. Maximize and marginalize working backward from the last decision in the network.

The *step* size for this algorithm is the number of large-valued instantiations that are computed.

2.2.5. Random

This algorithm is as simple as it sounds: the agent delays taking any action for a fixed period of time (the *step* size for this algorithm). It then selects an action at random and implements it. One might reason that this algorithm could be as good as anything

when the agent is permitted only extremely small increments of time, as the costs of deliberating might at some point outweigh the costs of taking random actions. Certainly this algorithm provides a lower bound on performance, since algorithms performing worse than this one are not worth considering.

2.3. Partially Observable Markov Decision Processes

Researchers have modeled action under uncertainty with *partially observable Markov decision processes* (POMDPs) for more than three decades, demonstrating the model's flexibility by the wide range of diagnostic and reactive tasks modeled. Strictly speaking, the system discussed in this dissertation may be more accurately modeled as a *cost-observable Markov decision process* [Bayer 99], but given that the models are polynomially transformable into each other I will restrict this discussion to the more generally discussed POMDP model.

A Markov decision process (MDP) consists of a tuple $<\mathfrak{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}>$, where $\mathfrak{S}$ is a set of possible states of the world or system being modeled, $\mathcal{A}$ is a set of agent actions, $\mathcal{T}$ is a mapping from $\mathfrak{S} \times \mathcal{A}$ into probability distributions over $\mathfrak{S}$ (i.e. a transition function which may be nondeterministic), and $\mathcal{R}: \mathfrak{S} \times \mathcal{A} \times \mathfrak{S} \rightarrow \mathbf{R}$ is a reward function stipulating the immediate real-valued reward given a transition from the first state to the second with action $\mathcal{A}$. Generally researchers are less concerned with immediate rewards than with the long term rewards generated from a plan of action that depends solely on the current state of the system. Such a plan is called a *policy*, and the problem posed by an MDP or POMDP is usually stated as one of finding the optimal policy.

An interesting property of MDPs is that under minimal assumptions there exists a stationary policy which is optimal in the sense that the total cost of performance under the policy is less than or equal to (or greater than, in terms of positive rewards) any other policy.

Straightforward methods exist for solving completely observable discrete-time MDPs. Linear programming can be applied to easily maximize expected reward.

Difficulties arise when we introduce partial observability. Here one still assumes the same MDP structure given above, but instead of allowing the decision maker to observe the system state before making an action choice, one adds a set of observations $\mathcal{O}$ such that one observation is made available to the decision maker after each state transition. The observation produced is correlated with the state transition, but does not generally allow us to completely determine the current state. The reward function typically is modified to include the set of observations in its domain.

While a discrete POMDP may be transformed into a continuous MDP [Astrom 65], this requires expansion from a finite to an infinite state space and results in an exponentially larger problem with prohibitively high costs for policy generation. Other techniques have been developed for continuous (belief) MDPs that improve performance in some instances. Value iteration generates an optimal policy by maximizing expected reward within t steps of the initial state. As t tends to infinity, it may be shown that the difference between the computed policy and the optimal policy goes to zero [Howard 60]. The problem of a continuous solution may be eliminated by an arbitrarily accurate approximation with a piecewise linear convex function [Smallwood 73]. Unfortunately, such solutions not only fail to guarantee polynomial time or space performance for

POMDPs, they also fail to guarantee that the optimal policy will be sufficiently small that the decision maker can store and use it. While some POMDP-specific algorithms guarantee a solution will be computed, models with even 100 states strain the limits of what is computable [Cassandra 97].

Some recent work attempts to generate approximate solutions quickly. The Witness algorithm constructs approximations to the optimal policy in conjunction with value iteration by seeking individual points where the current policy is inadequate and replacing each by a vector which improves performance in that region [Cassandra 94]. Preliminary tests of the Witness algorithm demonstrated convergence in approximately one day for the 256 state On-Line Maintenance Agent of this dissertation [D'Ambrosio 96b]. The authors of the Witness algorithm admit there are many limitations and have studied some improvements which may permit better scaling up [Littman 95b]. Other approaches have explored relaxing assumptions about discounted reward [Jaakola 94] or the form of the value function [Parr 95] with limited success.

Theoretical results suggest that previous approaches to working with partially observable stochastic domains have missed the mark. Many common policy improvement algorithms for MDPs exhibit exponential worse case complexity [Melekopoglou 94]. Computing an optimal policy for MDPs is NP-complete under all current formulations, and partially observable variants of these are PSPACE-complete [Papadimitriou 87]. A variety of policy existence problems for POMDPs are shown to be NP-Complete [Mundhenk 00]. Approximation may be of minimal help in many POMDP domains: optimal stationary policies can be ε-approximated for any $\varepsilon < 1$ if and only if NP = P [Lusena 98].

Some approaches to addressing these issues for certain classes of MDPs have been outlined [Littman 95], but it is not clear that such approximations will help in POMDP domains without further domain restrictions.

If we eschew policy approximation, there appear to be two viable alternatives. The first is problem reformulation. In many domains an alternate representation of the problem may result in significant simplification. Such alternate representations may be difficult to discover and integrate, however. The second alternative is to not compute policies at all. One form of this alternative is to reason dynamically in the environment. Dynamic reasoning also more readily permits change to one's model of the world, better supporting domains like robot navigation where the environment is expected to change over time. Dynamic reasoning is the approach I have taken here, and the one I believe promises the best and most immediate returns.

Why should we care? Because the POMDP model embraces many of the essential elements of domains currently studied in the AI literature. Facing the difficulties imposed by this model is necessary for the continued advance of the field.

2.4. Stochastic Automata Networks

Stochastic automata networks (SANs) are becoming an important modeling tool for the performance of parallel and distributed systems because they permit quantitative analysis of collections of components that infrequently interact [Fernandes 96]. While they also suffer from computational limitations, we will see later that they may be applied to yet another exciting domain: the modeling of agent actions.

SANs are an extension of Markov chain analysis, a widely used technique for modeling physical and economic systems with discrete state spaces. They are particularly appropriate when one desires to know the probability of being in a particular state after the system has run for a while, or the long term average time spent in different states in the space, though other important quantitative results are possible. I won't thoroughly introduce the mathematics of discrete-time Markov chains here: William Stewart has done a much better job than I could hope to [Stewart 94], and a reader unfamiliar with what follows should turn there for clarification.

A discrete-time finite state system may often be represented by a matrix of transition probabilities P where each p_{ij} represents the probability of transitioning to state j in the next time step given that the system is in state i. Such a *transition probability matrix* is said to have a *stationary distribution* if there exists some vector v such that $vP = v$. This stationary distribution may be interpreted as the percentage of time that the system will spend in each state ignoring an initial "start up" period.

Often one may generate many Markov chain descriptions from several sources, including stochastic Petri nets, queueing networks, or an explicit Markov state space model. If global behavior involving such components needs to be studied, SANs can sometimes provide the necessary machinery to combine and analyze these Markov chains. Usually the explicit transition probability matrix is not generated, since the number of states of the global system can quickly explode [Stewart 96]. However, the system modeled later in the text has only simple components, making explicit generation of the systemic transition probability matrix and its associated stationary distribution the preferable approach.

Given square transition probability matrices A_{mm} and B_{nn} representing two noninteracting discrete-time Markov chains, the transition probability matrix for the two-dimensional system incorporating both is simply the Kronecker tensor product C of the two matrices:

$$C = A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & a_{13}B & \cdots \\ a_{21}B & a_{22}B & a_{23}B & \cdots \\ a_{31}B & a_{32}B & a_{33}B & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

This matrix globally describes transitions that remain local to the component automata. Note that the dimensions of the new matrix are mn rows and mn columns.

When dealing with continuous time or interacting automata, one may also need to calculate the tensor sum D of two square matrices:

$$D = A \oplus B = A \otimes I_n + I_m \otimes B$$

Similar to the discrete-time case, the tensor sum of two noninteracting continuous-time Markov chains is the transition probability matrix for the global system.

When two stochastic automata interact, C and D are inadequate to describe the global system. Two kinds of interactions may be modeled: *functional transitions*, where the state of one automaton is a function of the states of another, and *synchronizing*

transitions, where the change of state of one automaton simultaneously forces a change of state in another. I will first address synchronizing transitions.

If one separates the local transitions from the synchronizing transitions, the global transition matrix may be described as a sum of tensor products. Further, it has been shown that in the continuous case the transition matrix may *always* be computed by first generating the global state transition matrix with a tensor sum without synchronizing events, and then adding two additional tensor products per synchronizing event. In general, then, the global transition matrix T^* may be written as:

$$T^* = \sum_{i=1}^{2S+N} \bigotimes_{j=1}^{N} T_i^{\,j}$$

where N stochastic automata with S synchronizing events are being combined into one global system [Stewart 94]. These results fundamentally carry over, with modifications, to the discrete time case [Stewart 00].

Functional transitions add an additional wrinkle to continuous Markov chains. While by themselves they do not change the structure of the global transition matrix, they may alter the transition rates. Generalized methods for dealing with this are more complex, and involve an extended tensor algebra that may be difficult to apply. They can become quite unmanageable when the number of transitions is large. When a transition is both functional and synchronizing, as some in this dissertation are, the general methods require significant mathematical machinery. However, small systems of discrete-time automata may be approached ad hoc, and that approach was preferable for the work I will discuss later. While computational difficulties exist, the reader should be aware that

there are more generally applicable methods which significantly broaden the utility of SANs applied to analysis of agent behavior.

3. The On-Line Maintenance Agent

3.1. Imposing Structure on Real-Time Inference

The previous chapter addressed alternate approaches to this dissertation, as well as algorithms represented in it. This chapter presents the On-Line Maintenance Agent (OLMA), which was developed over a period of several years by Bruce D'Ambrosio and students Tony Fountain, Caryl Westerberg, and Lothar Kaul at Oregon State University. I gathered all of my empirical data on the OLMA. While my additions and modifications were small in comparison to the complete OLMA, the OLMA is central to the dissertation and therefore deserves discussion apart from less intimately related work.

The OLMA was originally developed to explore compilation of reactive solutions and automatic construction of reactive controllers [Kaul 91, Westerberg 93]. Both of these theses describe learning components no longer used, as the OLMA has evolved into a testbed for studying inference algorithms and issues such as Incremental Probabilistic Inference [D'Ambrosio 93], value-drive diagnosis [D'Ambrosio 92], and comparisons of anytime decision algorithms [D'Ambrosio 96]. The remaining sections of this chapter present a more detailed hierarchical description of the On-Line Maintenance Agent.

3.2. The Task Level

Relative to earlier formulations of diagnosis, the OLMA is a unique tool for empirical explorations of scaling reactive controllers because it embodies several traits that more accurately mimic the real world:

- The OLMA is an embedded diagnostic domain comprising an *agent* and a *simulator* that share information but otherwise are treated as separate processes. This allows the agent to solve different problems, and different kinds of agents to solve the same problem.
- The simulator, for purposes of this dissertation, mimics a half-adder with several gate components, each of which may independently fail during a time step with some small probability.
- The agent may perform both repair and sensing actions on the simulator, and actions have costs.
- The agent also incurs costs for each time cycle the simulator fails to function properly, and it therefore has incentive to reason quickly and act in order to minimize costs.
- The performance metric is the agent's long-term costs for maintaining the simulator; the gold standard is minimum long-term cost.

The challenge for the agent is to accurately diagnose and repair the circuit in the simulator while fighting time constraints on deliberation. This is in contrast to frequent formulations of diagnosis as a static activity where an optimal or near-optimal action is chosen without regard to the costs imposed by the passage of time.

3.3. The Agent-Simulator Level

At this level the OLMA may be viewed as two processes: an agent and a simulator. The simulator proceeds in discrete time steps. At each time step of the simulator, information is sent from the simulator to the agent. The simulator presents the

agent with the inputs and outputs of the half-adder circuit during the last time step, and if the agent requested a probe point value from the last time step the value at that probe point is also passed to the agent.

The agent is also stepped, and while the steps of agent and simulator are synchronized the ratio of agent steps to simulator steps may be varied. At each time step of the agent, the agent sends an action to the simulator.

In the current implementation of the OLMA, the processes are not forked but are maintained through a control loop. Each process is allocated a small amount of computation time called a *CPU Quantum* that is a multiple of the process execution time recorded within the LISP interpreter used to run the OLMA. Since the simulator usually requires significantly less time to execute than the agent does to deliberate, the simulator is allocated a fixed CPU quantum, while the agent's CPU quantum may be varied by the user to simulate different processor speeds. Measurements of the agent's CPU quantum are referred to hereafter as *Quantized CPU Speed*. In most experiments with the OLMA, the agent's CPU quantum is varied as a power of two, since that results in regular significant jumps in perceived processor speed. The intent of this temporal structure is to allow the simulator to continue operating while the agent deliberates on its action choices.

An error in the structure of the current OLMA implementation makes the agent reset the simulator each time it posts an action. This means that whenever the agent's CPU quantum is larger than that of the simulator, the agent and simulator will be completely synchronized. Originally this simulator reset had been programmed into the OLMA as a computation-saving feature, but it had not been changed as both hardware

and the OLMA's purpose advanced. Discovering the effect of this error was one small part of explaining the data in the next two chapters.

As noted in [D'Ambrosio 92], the CPU quanta play an important role in determining interesting decision-theoretic behavior. If the agent has too much time to deliberate, it can find an optimum action over a short (several steps) horizon look-ahead problem. If the agent lacks sufficient deliberation time, its actions will be largely random. Since computational constraints require us to restrict computation to one-step look-ahead, we must account for the long-term effect of the agent's actions on the simulator. This is particularly important when one considers the effect of multiple faults, which may require several time steps for the agent to repair.

Fortunately, probe actions are largely resolved by one-step look-ahead, since the information content obtained is probabilistically summarized in a single advance. For replacement actions, D'Ambrosio chose to model replacement costs using an assumption of policy stability: if the agent chooses to not replace a component now, then it will make the same choice in future states, all other things being equal. The temporal extension of replacement decisions can then be modeled by using a multiplier for failure costs, which accounts for a failure to replace. This leads to two constraint equations:

$$mc_f \gg r$$ **Equation 3.1**

$$mpc_f \ll r$$ **Equation 3.2**

where:

- r is the cost of replacing one component in a time cycle
- p is the probability of component failure per time cycle

- c_f is the cost of a component failure per time cycle
- m is the multiplier for failure costs

We realize Equation 3.1 is necessary when we note that if the cost of replacement appears to be too large, the agent will never replace a component. Equation 3.2 accounts for the possibility of component failure, since if the expected cost of failure is too large then the agent will always replace components. Behavior at the margins of these equations has not been studied. Further details of this decision-theoretic formulation may be found in [D'Ambrosio 92].

3.4. The Component Level

This level describes the internal details of the simulator.

The simulator emulates a half-adder circuit with four gates, as shown below in Figure 3.1, where I1 and I2 are the inputs to the circuit, and P1 and P2 are probe points. Each gate is treated as a separate component that can be in one of four states during a cycle of the simulator: OKAY, STUCK_1, STUCK_0, and UNKNOWN. An OKAY gate functions as one would expect it to. When a gate is in state STUCK_1, it will output a 1 regardless of its input, and similarly STUCK_0 gates output a 0. When a gate is in the UNKNOWN state, it stochastically outputs either a 1 or a 0 independent of its input.

Since each gate is independent of the others, the circuit may have multiple faults. Also, with probability equal to the probability of a fault occurring in an OKAY gate, a gate in one of the fault states may return to normal operation. The failure probability was set to 0.003 per gate per cycle of the simulator for these experiments.

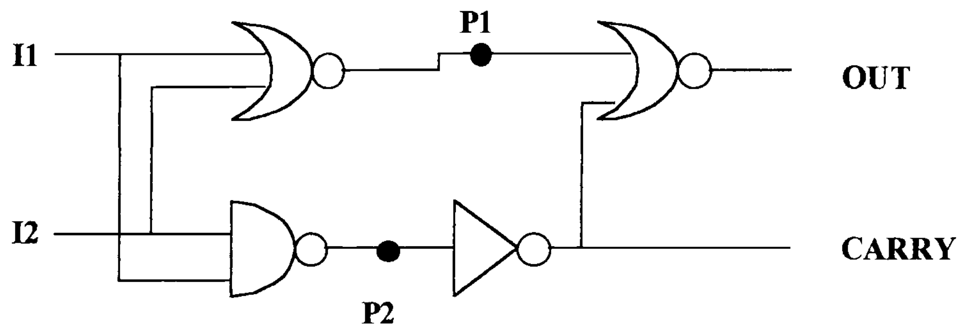

Figure 3.1: The Simulator's Half-Adder Circuit.

If a gate is in one of the fault states during a simulator time cycle, cost is incurred for that gate's failure and added to the total cost recorded for the agent. For the experiments in this dissertation, the failure cost was 1 per gate per time cycle of the simulator. The agent is not informed of the costs thus incurred, as the simulator only shares input/output pairs and probe values with the agent. The simulator is viewed as having only one probe that may be moved between the two possible probe points, limiting the agent to requesting only one probe value.

3.5. The Action Level

This level describes the internal behavior of the agent.

The agent is a reasoning entity charged with maintaining the proper function of the simulator. The agent maintains a belief model of the simulator, updates the model in accordance with observations obtained from the simulator, and deliberates using that model in order to choose an action to send to the simulator.

The belief model at a time slice may be viewed as collections of states S_j, collections of observations O_j, decisions Dj, and a value node. Each time slice requires information from a previous time slice to act as priors on the belief states of the gates (in the first time slice, gate states of OKAY are assumed in the priors, and the "observations" from that slice are normal). The current time slice has a single decision node. The temporal projection is accomplished through a single look-ahead time slice representing the extent of the current decision. It contains both a collection of state nodes and a decision node for the decision immediately following the current one to be made, as well as a collection of nodes representing the states to come after the next decision. Current and future decision and nodes and future state nodes affect the value node for the network. An abstracted arbitrary time slice is depicted in Figure 3.2; a more detailed representation of a time slice showing the individual gate representations is given in

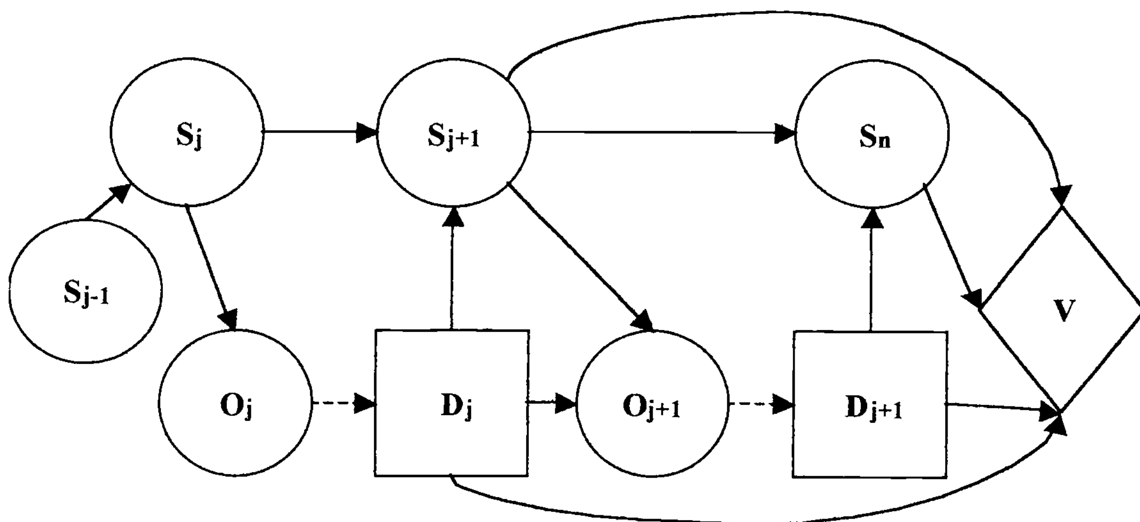

Figure 3.2: Abstraction Basis For an Agent Decision.

Appendix A. The arcs from the observation nodes to decision nodes are dashed because they are informational rather than causal links.

The agent must regularly update this model to account for completion of a decision cycle and consequent action taken. To do this, the agent effectively rolls the time slice one time unit forward. The current time slice state nodes will become the previous time slice by replacing the previous stage with the factored joint probability distribution across component states after the current decision is made. The approach for this can vary slightly for each algorithm, but generally exact inference is used to estimate these updated prior probabilities. The future time slice is simply replaced by a new future time slice with static prior distributions and a new value node. By utilizing a summarized past state and limited projection for future states, the size and inference time for the network remain essentially constant during a run of the simulation.

Once inference has been performed, the agent must select an action. There are seven possible actions when the agent is configured for the half-adder: NOTHING, which has no effect on the simulator; four REPLACE actions, one for each gate in the half-adder, which cause the simulator to return the state of a gate to OKAY if it is in one of its three broken states; and two PROBE actions, one for each probe point in the circuit. The agent chooses one of these possible actions each time it completes a decision cycle. REPLACE and PROBE actions incur costs of 10 per gate replaced and 1 per probe, respectively. Doing NOTHING has zero cost. The agent's model of costs incurred by the simulator includes the temporal extension of the cost of a failure, which is three times the failure costs per gate per time cycle, or 30.

4. Explorations with Several Inference Algorithms

4.1. Experimental Goals

The broad goal of these experiments, as stated earlier, was to better understand and perhaps improve upon existing anytime inference algorithms. The goal of the experiments in this chapter was to explore the performance of several algorithms and attempt to confirm several hypotheses from the literature and also from the creation of the testbed itself:

- Druzdzel hypothesized that one could make good decisions looking at only the few largest terms of the full joint probability distribution [Druzdzel 94]. He reasoned that since failures, diseases, and similar events were usually rare, diagnosis upon such failures normally would result in few explanations for observations. D'Ambrosio and I wished to confirm that the testbed exhibited such behavior.
- A related question is what happens when the agent is given more time, and can compute perhaps a few more terms. Does computation improve with more terms? Does this improve overall utility? It is not obvious whether the extra time spent computing will improve decision quality sufficiently to overcome the costs of not taking action now. It is also not clear that the optimum amount of time that the agent should use for making a decision is always the amount available between steps of the simulator; it may be that the agent's costs for deliberation are less than the cost of inaction.

- Given that the agent can be set to spend more or less time on decision making, is there a smooth tradeoff between time devoted to reasoning and the quality of decisions, or is there a brittle edge where costs suddenly spiral out of control?
- Does the optimum amount of computation vary with the CPU speed of the agent?

4.2. Method

To investigate these hypotheses, a few incremental algorithms were chosen. An algorithm we call Kappa-Reduced was suggested by Moises Goldszmidt's work on the Kappa Calculus [Goldszmidt 95]; and D-IPI was already available, having been developed some years earlier at Oregon State University [D'Ambrosio 93]. A stochastic simulation algorithm was not included in these early experiments due to its reputation for slow convergence. D'Ambrosio and I included two algorithms to help benchmark performance: SPI exact inference (hereafter called Exact) and random choice of action (hereafter referred to as Random) provided lower bounds on performance, since if an anytime algorithm cannot outperform both of these it is not worth serious consideration.

After collecting our initial data, we felt the Kappa-reduced algorithm had performed poorly so the Kappa-reduced algorithm was revised to create the Posterior Kappa-reduced algorithm. These algorithms are discussed in Chapter 2.

Choice of a performance metric was crucial to algorithm evaluation. Typical methods that had been used to evaluate reasoning algorithms included running the algorithms on a common set of decision problems, and collecting data on randomly

generated graphs. Prior to my involvement, D'Ambrosio had already realized the difficulty translating standard performance metrics into something which a program designer could use to make decisions about which algorithm was most appropriate. What does a typical influence diagram we would like to do reasoning on look like? How do I compare results on one set of influence diagrams to expected performance on another set of influence diagrams that I am interested in? The difficulties are even more pronounced with incremental algorithms specifically designed to provide best performance only under tight time constraints.

We elected to measure performance by using the cost per unit failure for a long run of the system. While the cost per unit failure metric does not address differences in influence diagrams, when used in conjunction with a realistic testbed like the On-Line Maintenance Agent it does provide a reasonable performance measure for our incremental algorithms. Since our efforts here were exploratory rather than a full performance comparison of the algorithms, we felt there was less need for coverage of a wide range of influence diagrams. What we did need was a means of evaluating the effects of different reasoning algorithms on decision-making processes.

Designing the data collection runs was nontrivial. Several parameters need to be considered when collecting performance data using the OLMA. Random requires a trivial amount of deliberation time, of course, so there is no tuning required. Exact is nonincremental, so it must run to completion. Allowing more time than necessary for making decisions will increase the cost per unit failure, since the only effect on cost is to delay repairs for one or more decision cycles. Again it is not clear that giving the agent

more time will result in a completely smooth increase in costs, but we hoped the experimental data would answer that question.

Our most advanced algorithms must be tuned for best performance. The incremental algorithms all may vary the number of incrementally refined results computed. Observation suggests that with any incremental algorithm, additional useful computation tends to improve the cost per unit failure metric. But since the agent and simulator can be set to effectively run at different speeds, we might expect that the optimal amount of computation might vary with the quantized CPU speed. In these early experiments the exact relationship expected was unclear. Thus it was necessary to collect data varying both quantized CPU speed and the number of steps of the incremental algorithms.

But fixing the steps under the On-Line Maintenance Agent required additional labor: since the agent was forecasting future results by projecting the long-term effect of repair and non-repair actions, the agent needed to know how many simulation cycles would transpire per decision cycle. This required setting an additional parameter, the *decision-time*, so that the agent could make an accurate estimate. Some preliminary runs were enough to convince Bruce D'Ambrosio and me that none of the algorithms were particularly sensitive to the exact value of this parameter. Nevertheless, I took pains to check the average decision time for different quantum/step combinations by using preliminary runs, and select the next greatest integer value over the average for the data runs. Preliminary runs used to determine decision-time settings were typically only 100 decision cycles in length.

We evaluated the preliminary runs to judge how long each run of the algorithms would have to be to collect the data. Since a number of parameters were being varied, we had hoped that 100 decision cycles would be sufficient to collect reasonable performance data, but it quickly became apparent that at some quantum/step combinations the simulator would not generate an adequate number of faults in the system in 100 decision cycles. Consequently our metric, the cost per unit failure, would be based on a very small number of failures. To address this deficiency, we elected to increase the number of decision cycles to 500 for each data run.

Additional constraints resulted from a lack of uniform computational resources. Several workstations were available in the department that ran the Sun Solaris operating system and the necessary LISP interpreter, but processor speeds and memory configurations differed. Different processor speeds directly affect performance since time allotted to the agent and simulator is measured in milliseconds of CPU time. There may also have been communication speed variations with the main file server. To address speed issues, I used two strategies: some algorithms were run exclusively on a single machine, and other were run on multiple machines where differences could be accounted for by simple math. The latter case applied only to the Kappa-reduced algorithm, which was run on Oregon State University's Sparc machines known as Hume (Sparc 1), Godel (Sparc 1), and Alhazen (Sparc 2). Since Alhazen ran at half of the speed of Godel and Hume, the decision-time settings of the Alhazen runs were different, but I observed no discernable differences in the cost-per-unit-failure measurements (which is what we expected). D-IPI and SPI Exact runs were collected on a Sparc Classic known locally as Ptolemy.

The LISP interpreter allows one to fix the starting value of the seed for the pseudorandom number generator. Since both the agent and the simulator are using the pseudorandom number generator while performing their tasks, and actual timing between the two subprocesses can vary, it is probable that the sequence will diverge in time. To ensure that common subsequences were less likely, I manually force the pseudorandom number generator to produce an arbitrary number (between 1 and 9999 inclusive) of pseudorandom numbers, thus reducing the likelihood of a commonality between runs. Ideally separate seed values would be kept for the agent and simulator, both to aid in debugging and to allow more careful study of the components, but I saw little need for it in my own experiments and consequently decided not to implement this change.

When reviewing the data for this dissertation, I noted some discrepancies between data sets I had collected and those published by Bruce D'Ambrosio and me [D'Ambrosio and Burgess 1996]. Since I had kept careful records of what I had done, I reviewed my own data and methods carefully. I noted that for data sets with D-IPI and Posterior Kappa-reduced, there were large differences between my data and those published. For these same algorithms, I had done data collection sets over 500 decision cycles, but had used 100 decision cycle runs to gather data necessary for determining the decision-time values to use for the actual data runs. These shorter runs had much lower cost-per-unit-failure values with these two algorithms. Since D'Ambrosio did not keep track of details of his methodology, and we were collecting data rapidly in order to meet the paper deadline, I suspect that D'Ambrosio's data is based primarily on shorter 100 decision cycle runs.

If the data are accurate, there may be a flaw in either the testbed or in the implementation of these algorithms that accounts for this "start-up" effect. Careful checking of the code did not reveal a possible source for such an error. Further tests with shorter runs of the Random algorithm (which is thoroughly studied in later chapters) failed to turn up further examples of this effect. Longer runs of 500 to 1000 decision cycles appear to be more stable. Consequently I have chosen to present my own data in the discussions that follow.

The data originally presented for the Random algorithm in the UAI-96 paper are clearly in error. We had collected a single data point and projected its behavior for different quantized CPU speeds on the belief that the quantized CPU speed did not affect the cost per unit failure. Research I present in later chapters shows that this is incorrect because it does not correctly account for effects attributable to the structure of the OLMA. An important result of these investigations is the reminder that effects attributable to the simulator's structure must be understood and clearly separated from the behavior of the algorithms being investigated.

Actual data values for these algorithms are presented in the tables of Appendix A. While I do not report the raw data values in most cases, I report both D'Ambrosio's data and my own, the latter forming the basis for the results which follow.

4.3. Results

Figure 4.1 summarizes the observed cost-per-unit-failure results with the Kappa-reduced Algorithm.

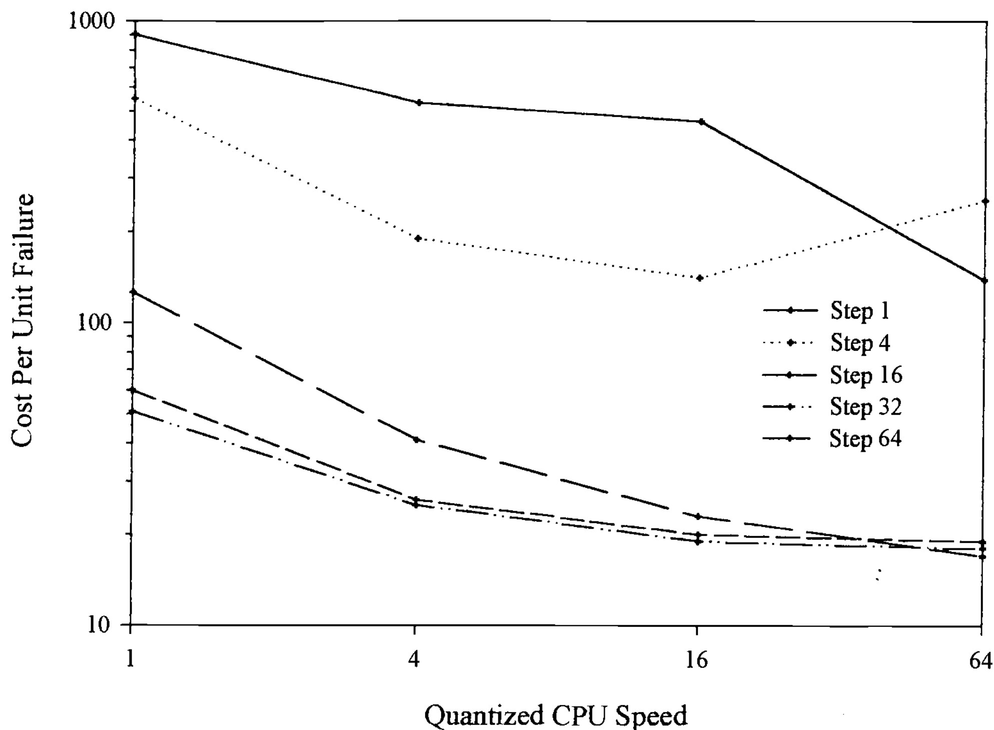

Figure 4.1: Cost Per Unit Failure of the Kappa-reduced Algorithm Across Step Size

In our UAI-96 paper [D'Ambrosio and Burgess 1996], we dismissed the Kappa-reduced algorithm as noncompetitive, but that conclusion may not be completely justifiable. From my data sets it appears that the Kappa-reduced algorithm is reasonably competitive at step values 16 and 32, but somewhat brittle with respect to step size. I now wish I had collected data at the neighboring step value 8.

The results for the D-IPI algorithm are presented in Figure 4.2. There was insufficient time before the publication of the UAI-96 paper to collect sufficient runs for averaging, so all but a few data points are plotted from a single run. Performance at step value 4 looks modestly promising, but as with the Kappa-reduced algorithm a degree of brittleness with respect to step size is notable.

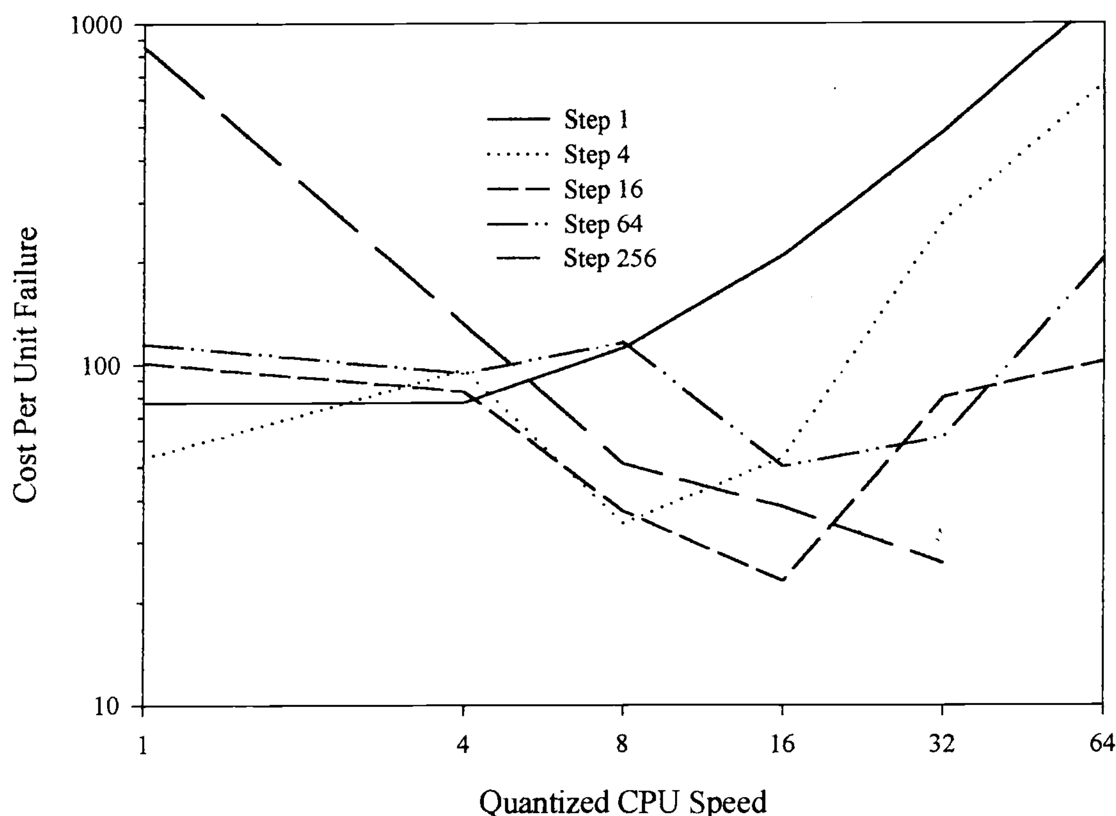

Figure 4.2: Cost Per Unit Failure of the D-IPI Algorithm Across Step Size

Such brittleness can cut both ways: while tuning the algorithm is easier in the sense that it is easy to find a step size with superior performance, one must be careful to do so in order to guarantee that the agent will perform satisfactorily.

The results for the Posterior Kappa-reduced algorithm are presented in Figure 4.3. Here we have a remarkable change: the brittleness of the previous algorithms is gone, and performance is consistently good across step sizes. Not only that, but performance appears to be better on average than in either of the previous algorithms precisely where

it is most needed: when the quantized CPU speed is at its smallest. Note that several step values are "best" at different quantized CPU speeds.

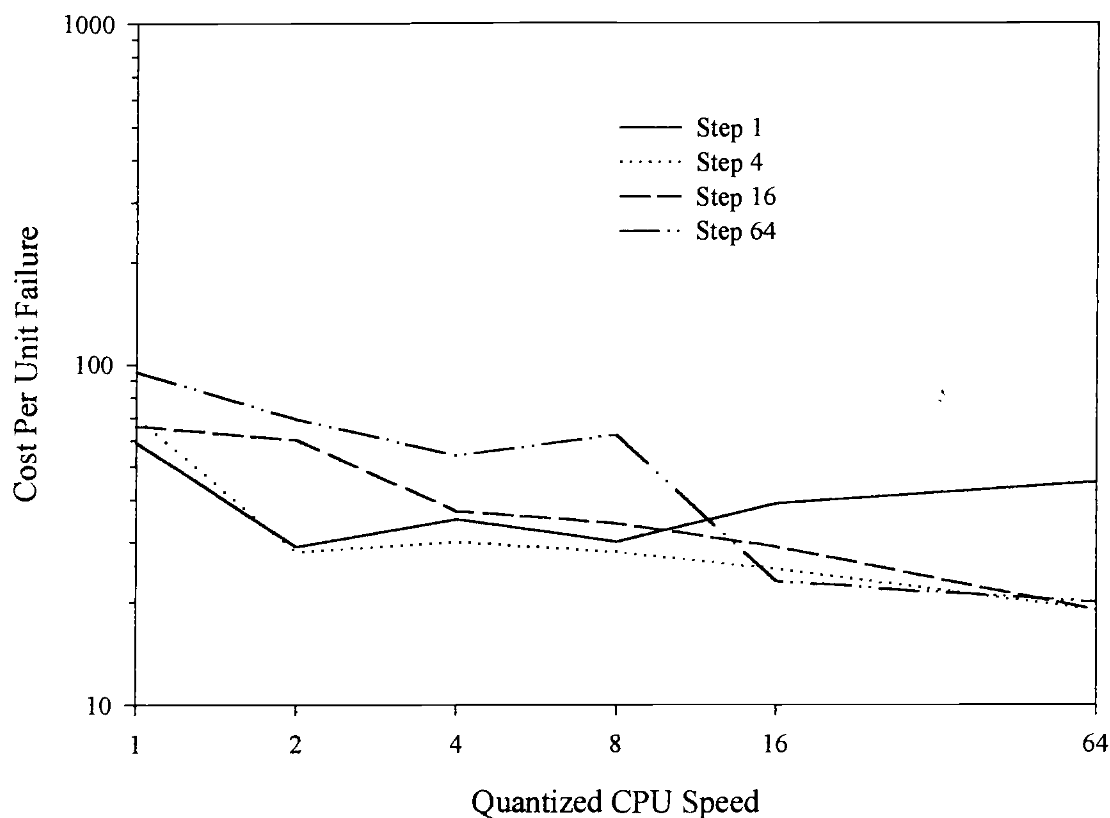

Figure 4.3: Cost Per Unit Failure of the Posterior Kappa-reduced Algorithm Across Step Size

One question we may ask now is how these algorithms compare when they are at their best. That is, if we choose step sizes for combinations of algorithm and quantized CPU speed, what is the best performance of each algorithm, and how do they compare with the Exact (SPI) algorithm? (Though the UAI-96 paper also compares these with the Random algorithm, I will reserve discussion of this until a later chapter.)

The best average performances for the anytime algorithms and for SPI Exact are presented in Table 4.1 and Figure 4.4. Empty positions in Table 4.1 simply reflect data that I was unable to collect due to time constraints. Note an important difference with the UAI-96 paper: I present *average* performance data for SPI Exact, not the best single

	Quantized CPU Speed						
Algorithm	**1**	**2**	**4**	**8**	**16**	**32**	**64**
Exact	**145**		**46**		**24**		**20**
D-IPI	**53**		**77**	**34**	**23**	**26**	**102**
Kappa-reduced	**51**		**25**		**19**		**18**
Posterior Kappa-reduced	**59**	**28**	**30**	**28**	**23**		**19**

Table 4.1: Comparison of Best Average Performance of Several Anytime Algorithms and Exact Inference

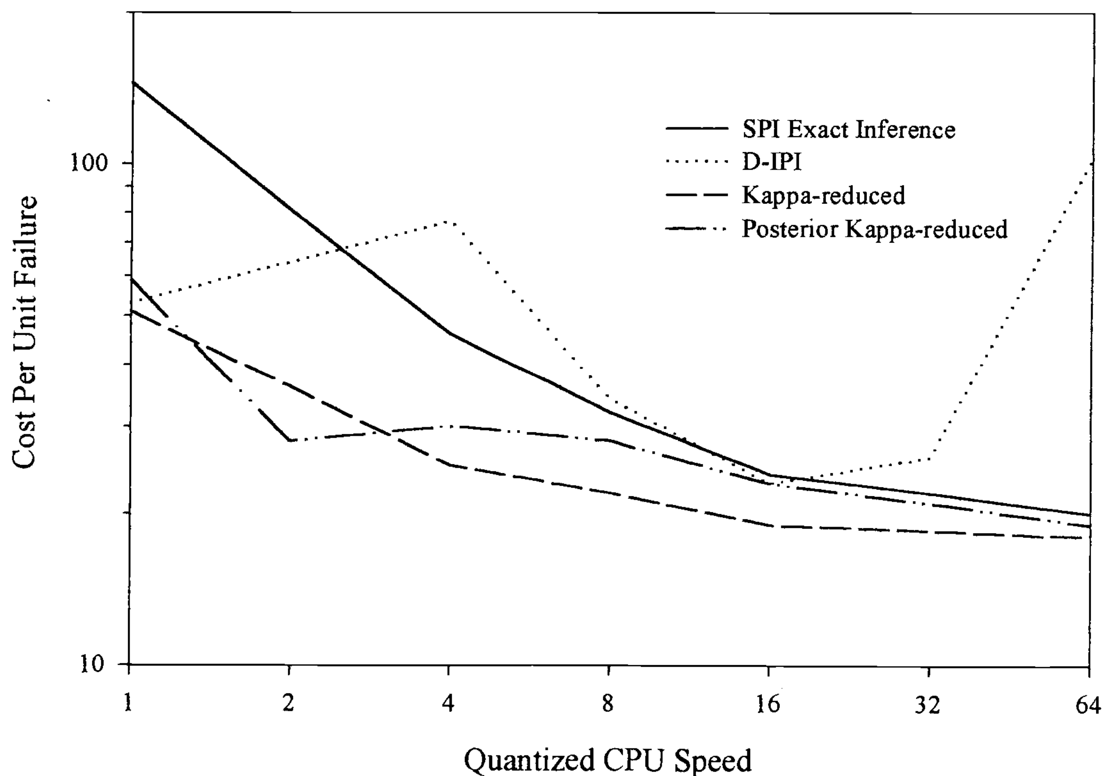

Figure 4.4: Comparison of the Best Average Performance of Several Anytime Algorithms and Exact Inference

performance overall. I also present the best average performance for all of the anytime algorithms. I feel this gives a better comparison of the strengths and weaknesses of the algorithms since variance becomes less of a factor (though it may be a factor for some values of the D-IPI algorithm presented here) and since averages are compared with averages. To generate the graph, missing data had to be replaced with the averages of values before and after in Table 4.1, though this may not be obvious at first glance.

While one would expect smoother curves in Figure 4.4, it is best to keep in mind that the averages are of only one to three data points for all data presented, and further that the individual data points could be more stable if each individual run was carried to a greater number of decision cycles.

4.4. Discussion

All of our anytime algorithms followed two expected trends:

1. Best average agent costs generally decrease when the agent has more CPU time.
2. The number of steps required to obtain best average performance generally increases as more CPU time is available.

These results appear to confirm our hypothesis that the anytime algorithms are effective at trading CPU resources for average quality of decision over a long run of the simulator. Further, the results justify our interest in such algorithms as part of resource-bounded inference.

A number of interesting questions arise from examining the results. First I note that the Kappa algorithms generally perform well in this domain. One might wonder

what properties make an algorithm appropriate for a domain, and I take up this question in the following chapter. One also has to wonder how the change from *a priori* to *a posteriori* estimation reduces variance as it does in the Posterior Kappa-reduced algorithm. I hypothesize that the Posterior Kappa-reduced algorithm more accurately focuses on states most likely associated with the evidence, and consequently exhibits less brittle behavior. Since some algorithms appear to require careful tuning for optimum performance, is the need for such tuning predictable and manageable?

Secondly, some anytime algorithms appear to perform quite well with very small amounts of computation. Are there characteristics of the problem chosen which make it amenable to anytime inference? Do parameters of the problem like failure probabilities, replacement costs, or inspection costs have much bearing on the amount of computation required? Part of the answer may lie in the benign nature of the problem: there are no dramatic costs for missteps, though it is notable that failure to repair a broken gate quickly results in mounting costs. Further study is necessary to answer these questions.

I performed a few manual examinations of the maximum subjective expected utility using the SPI Exact algorithm. What I discovered was startling: differences between the optimal action and the next best choice often differ by less than .01 and sometimes by less than .001 (keeping in mind that MSEU is expressed like probability mass in the OLMA, and total MSEU across the seven possible actions sums to 1.0). This is particularly true for states where no gate is broken, so an algorithm that is slightly biased towards inaction may generally perform better than one that isn't. It may also explain the "start-up" effect that differentiates short and long runs of the simulation under some algorithms, assuming that those algorithms have this bias and the simulator

generates fewer faults during its early stages; I have been unable to locate such a flaw in the simulator, but have not had the opportunity to test it thoroughly.

Lastly, the anytime algorithms generally outperform exact inference across the entire range of our experiments. So when does Random outperform Exact? Do the anytime algorithms outperform Random as we expected? What part of the performances of these algorithms should be attributed to the algorithms and what should be attributed to the testbed?

4.5. Related Results

In addition to the issues of scale related to parameters of the OLMA, there are three other scale issues worthy of discussion here:

1. How do our results scale with increasing problem size?
2. How do our results scale with increasing look-ahead (number of decision stages)?
3. How do our results scale with increasing numbers of internal states (requiring extended memory of the past)?

D'Ambrosio has begun to address the first two issues through other experiments reported in the UAI-96 paper. Experiments conducted on the OLMA the D-IPI algorithm was tested with circuits having one to sixteen gates. The time required to evaluate decisions or estimate posterior distributions grew only slightly greater than linearly with the number of gates.

To address the second issue, D'Ambrosio studied a different decision problem where both the number of computation steps and the number of decision stages were varied. Surprisingly, the D-IPI algorithm outperformed the Posterior Kappa-reduced, and

was able to search quite deeply while doing so, while the Posterior Kappa-reduced became intractable at depth 3. It is hypothesized that the result is due to the static nature of the domain restrictions of Posterior Kappa-reduced, but further experiments are necessary to demonstrate this.

I have designed an implementation of the master-slave flip-flop circuit to address the third issue. Since inputs from the last cycle of the simulator will produce the outputs of the current cycle, the agent must maintain a representation of the previous state and use current evidence to deduce what state the circuit was previously in. Assuming the agent's model of the simulator uses the obvious technique of representing a past state by using a past time stage, the agent may be required to extend any particular number of stages into the past by a simulator comprising that many flip-flops chained together. I have not had the chance to implement this circuit in the OLMA, as it requires changes to the OLMA's structure that do not appear to be justified by the impact of the results: while problems with internal state are of interest in the POMDP community, there is no obvious reason to suppose that the incremental inference problem posed by extending several stages into the past is significantly different from extending several stages into the future.

4.6. Conclusions

While it can be difficult to conclude much from a single testbed configuration, these early explorations confirm a number of important hypotheses:

- Kurtosis, or focus, may play an important role in the performance of these anytime inference algorithms. The experiments here are insufficient to reveal the enabling conditions for this, but they are sufficient to

demonstrate that the basis for trading computation time for quality of decisions is sound for some domains.

- Further, the experiments demonstrate that when the agent is given more time to make decisions, the quality of those decisions can be improved in an apparently fairly regular and smooth way.
- The optimum amount of computation for the agent varied not only with the CPU speed of the agent, but with the particular algorithm chosen.
- Some algorithms require significant tuning to obtain optimal performance, while others may perform well across a wide range of step sizes; estimating MSEU after propagating the effects of evidence for faults through the influence diagram appears to enable this non-brittle behavior.

In addition, a number of important questions are raised by these experiments:

- A number of experimental design issues are exposed: how can one fairly compare two algorithms, or even decide which algorithm is most appropriate for a new domain when so many variables (influence diagram structure and size, distribution kurtosis, and simulation parameters, to name a few) must be dealt with?
- How can one separate the behavior of the algorithms from that of the simulation's structure?
- Exact inference provides something of a lower bound on performance, but clearly an upper bound must also exist. Are the algorithms pushing close to this upper bound? How can we know?

The remaining chapters attempt to address some of these issues.

5. Some Experiments on Kurtosis

5.1. Experimental Goals

In our early explorations [D'Ambrosio and Burgess 1996], D'Ambrosio and I hypothesized that the distributions hidden in the influence diagrams contributed substantially to the performance of the anytime algorithms, and we pointed to earlier work [Druzdzel 1994] to support this claim. In those experiments Posterior Kappa-reduced significantly outperformed both D-IPI and Kappa-reduced, and we attributed the difference in performance on the ability of Posterior Kappa-reduced to better focus on portions of the search space where probability mass was likely to lie.

This experiment asks whether the kurtosis of the distributions implied by low-probability faults can be exploited by high kurtosis in sampling those distributions. In other words, does peaked sampling account for a significant portion of the performance of these algorithms? If this is the case, we know that "good" algorithms should possess this property in domains similar to the OLMA.

Please note that the "kurtosis" values discussed here are *not* equivalent to the more formal statistical use of that term, but represent my own informal attempts to quantify the peakedness of the sampling distributions.

5.2. Method

It is obviously insufficient to compare the algorithms already examined to answer this question: they differ on two many particulars for proper isolation of the kurtosis

property. Thus I had to design a new algorithm where the peaked sampling could be regulated as a property of that algorithm *without altering its other behaviors.*

Backward simulation provided a relatively simple algorithm, but how might one modify it to sample high probability mass zones of the distributions more heavily? Consider Table 5.1, which illustrates the effects of squaring and renormalizing a probability distribution. The variable A may be in states 1, 2, and 3 with the probabilities 0.6, 0.25 and 0.15 respectively. If we square each of these values, we obtain the values in the following row, which cannot constitute a probability distribution since they do not sum to one. But the values may easily be renormalized to generate the third row, which is a new probability distribution associating different values with each state of variable A.

	P(A=1)	**P(A=2)**	**P(A=3)**	**ΣP(A=i)**
Original probability distribution:	0.6	0.25	0.15	1.0
Elements above squared:	0.36	0.0625	0.0225	0.445
Previous row renormalized:	0.8+	0.14+	0.05+	1.0

Table 5.1: An Example of Peaking a Distribution with Logic Sampling

We notice immediately that if we sample according to the renormalized squaring instead of the original probability distribution, more samples will be allocated to state 1 at the expense of states 2 and 3. Such a sample does not give a true picture of the probability distribution, but this can be fixed: if after sampling 100 times we note that 81 samples were allocated to state 1, then we can recover an estimate of the true distribution for state 1 by multiplying by 0.6/0.8, the ratio of the original distribution to the peaked

distribution for that state. This yields an estimate of 60.75 samples for state 1, or an estimated probability of 0.6075 for that state.

By applying this peaking technique to backward simulation, we still obtain an accurate estimate of the probability distributions but with more samples than usual allocated to the high probability mass zones. My hope was that this would give a more accurate picture of these critical zones and thus improve the performance of stochastic simulation when only a small number of samples can be allocated. Of course, one is not limited to taking the square of the probability values, and I have termed the power to which the values are raised the *kurtosis value* for that sampling.

I decided to try a number of different kurtosis values to see if "more is better." I felt that in the limit all samples would be allocated to just the highest probability mass(es) in the distribution, which gives a poor estimate of the complete distribution. Thus if peaking improves stochastic simulation with a small number of samples, it was also likely that it did so only within a range of kurtosis values that depended somehow on the total number of samples. I decided also to test fractional kurtosis values, since those should exhibit the opposite effect of peaking with values greater than one.

The care I took when collecting data in Chapter 4 also applies here, so I won't reiterate details. But for careful comparative data, I decided further steps were needed to ensure that I could compare the performance of peaked and unpeaked sampling.

I realized that stability of the data runs was crucial. The number of failures exhibited for some combinations of quantum and step size had been too small for a few data points collected in Chapter 4. I decided to double the number of decision cycles for each run, meaning that each run would now be 1000 decision cycles long.

Secondly I noted that some statistical analysis of the results would be necessary. From the data collecting I performed for Chapter 4, I had learned that a simple string of experiments would require several months of CPU time and much babysitting of the processes in the event that they crashed. An upgrade of the LISP compiler appeared to improve performance with regard to jobs crashing, but how to collect sufficient data? Oregon State University possessed several parallel computers, including a 16-node Meiko CS-2 locally known as Shark. Unfortunately, Shark's message routing hardware failed as I was about to begin my data collection! I opted to conduct my experiments on an 8-way Sun™ Enterprise™ 4000 Server known locally as Trek.

I could not use Trek to the exclusion of others, but could run four or five processes fairly continuously. Since the operating system and hardware would be the same for all jobs run, I could eliminate those as variables of concern. But this still left me far short of the capacity necessary to generate sufficient data points for common tests that compare sample means.

Consulting with a statistics graduate student yielded naught. I queried a number of pharmacy and statistics faculty until the name Wilcoxen was mentioned by one. In a statistics text for scientists and engineers I then found the Wilcoxen rank-sum test, a test applicable for comparing the means of sets of samples when the number of samples per set is quite small. I now needed potentially as few as four samples per kurtosis value per quantum.

But how many would I *actually* need? This was uncertain. I chose to attempt to collect four samples per kurtosis value per quantum, and hoped that would be enough since I lacked both the knowledge needed to better estimate the number of samples

required and the time to collect more samples than this. I chose kurtosis values of 0.125, 0.5, 1, 2, 8, and 32. I chose to do the run at 32 expecting that performance might drop off with such a large kurtosis value, as explained above. Total time to collect this data exceeded six man months and four CPU years.

I collected data at quantized CPU speeds of 1, 2, 4, 8, 16, 32, 64, and 128 initially, but for kurtosis values 1 and 2 also collected data at 1028. I had observed some unusual shapes in the graphs, and hoped the additional data might reveal what the limiting process looked like.

5.3. Results

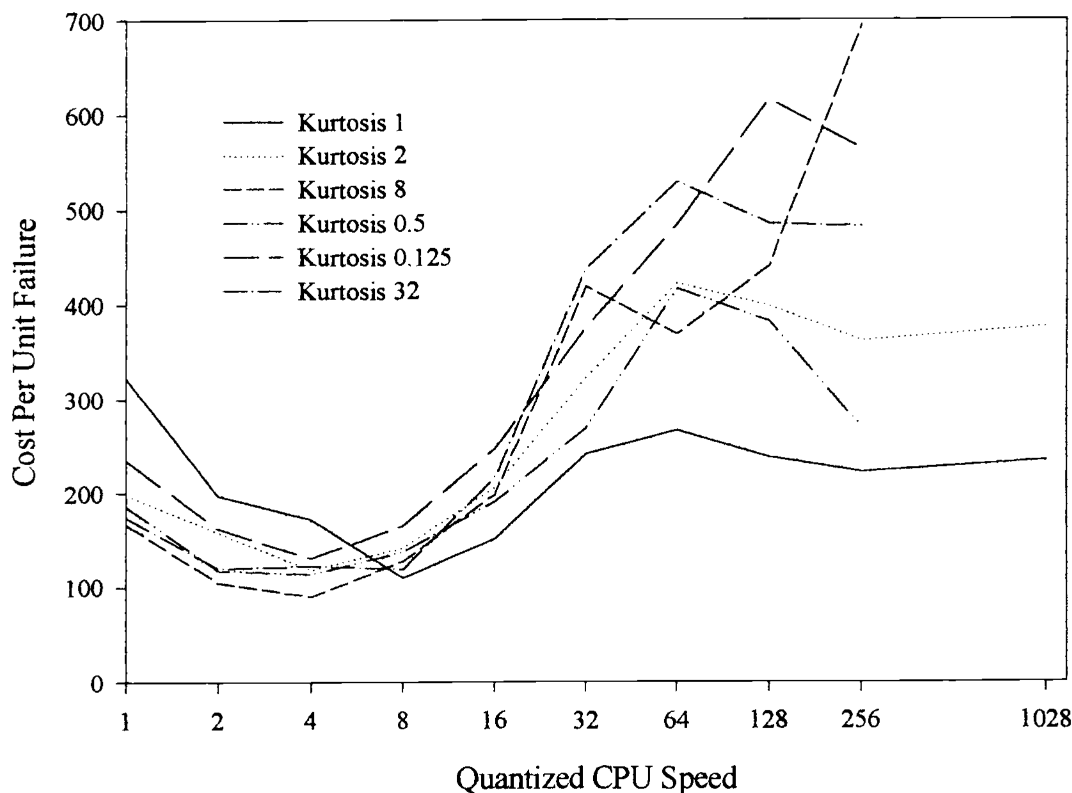

Figure 5.1: Average Performances of Backward Simulation with 1000 Samples at Different Kurtosis Values

The results for quantized CPU speed 1000 are summarized in Figures 5.1 and 5.2. The raw data from which this is derived may be found in Appendix B.

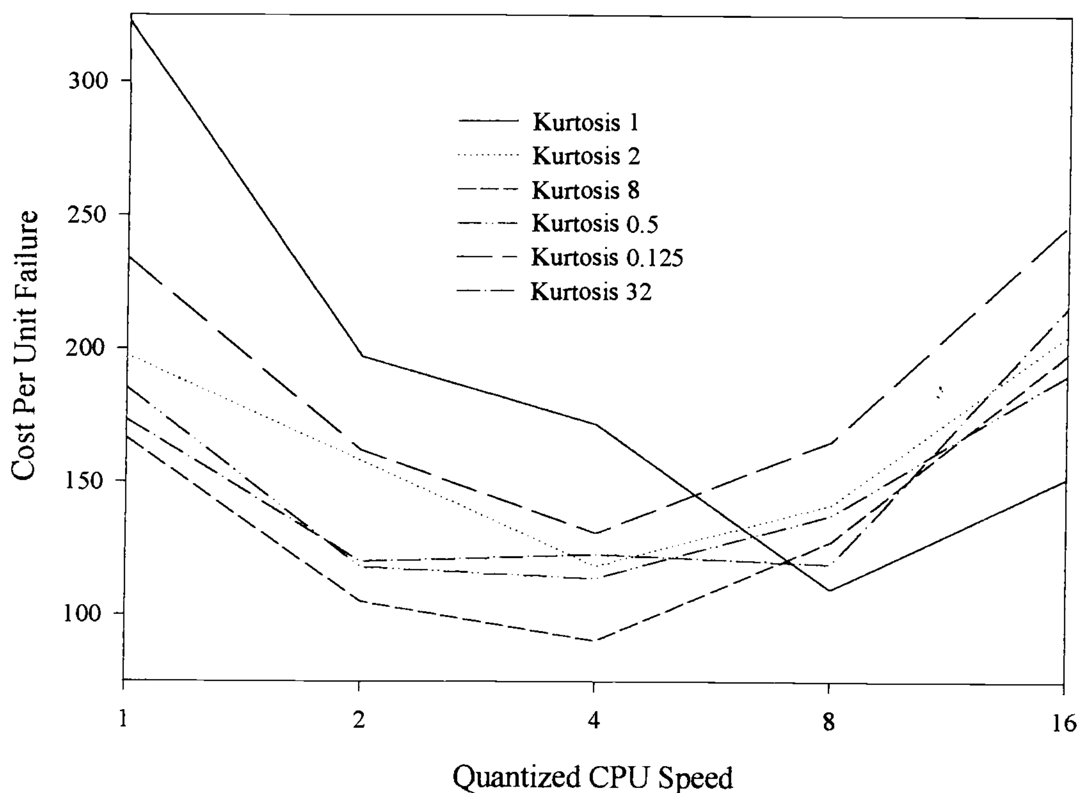

Figure 5.2: Critical Region of the Averaged Performance Data for the Backward Simulation Algorithm with 1000 Samples at Different Kurtosis Values

Studying these graphs carefully, I concluded there were three regions of interest. The first is the downward slope on the left where cost per unit failure is improving. The second is a rise in the cost per unit failure in the middle of the graph. But then performance seems to level off on the right third of the graph. For kurtosis 1 and 2 I carried the graph out further to quantum 1028 to make sure that this was not a local phenomenon, and the graph suggests it is not. Thus for some reason the OLMA's

performance became almost fixed after a particular quantum value; the performance within that rightmost region is determined primarily by the kurtosis value.

This rightmost region appeared to violate some of our assumptions about the performance of the OLMA. If the agent is running faster and faster, then eventually the agent should reach a point of diminishing returns where the extra effort to reason further incurs costs that exceed the benefits of reducing the time to repair. This becomes most apparent when we consider the SPI Exact algorithm. One interpretation of Exact is that once an exact solution has been computed, further time spent reasoning is wasted.

The most important point for each kurtosis value is where the cost per unit failure is minimum. I note two interesting phenomena in the graph. First, some data minima show performance differences with respect to the baseline, Kurtosis 1. The minimum value for Kurtosis 8 is lower than the minimum for Kurtosis 1. Applying the Wilcoxen rank-sum test for the data establishing these averages I found that the difference statistically significant at the level 0.025 (Table A.17 in [Walpole et al. 1985] was used in making this determination). The minimum value for Kurtosis 0.125 is higher than the minimum for Kurtosis 1 at the same level of significance. The minima for Kurtosis 2 and Kurtosis 0.5 could not be statistically separated from the Kurtosis 1 minimum with the given data. The minimum for Kurtosis 32 also could not be separated statistically from that of Kurtosis 1, but one data point in the sample set for the minimum of Kurtosis 32 is an extreme outlier that I elected not to discard since the number of simulator faults appeared to be sufficient to permit it. Ignoring that outlier, Kurtosis 32 would also give performance superior to Kurtosis 1.

The second phenomenon worth noting in the graph is that all peaked samples appear to have their minima shifted toward lower quantized CPU values. This is useful in one sense: minima that occur at lower quantized CPU speeds yield better performance precisely where the agent is pressed hardest under the time/quality tradeoff. Without a better model of the OLMA, it is difficult to say more about what this means.

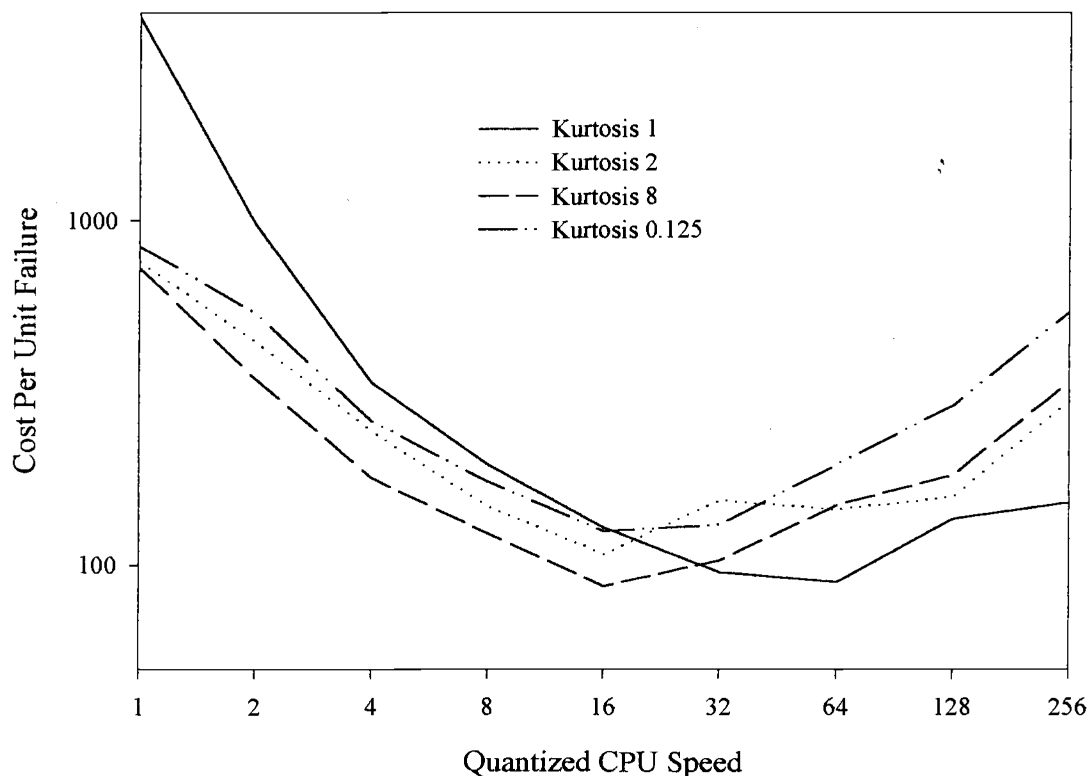

Figure 5.3: Average Performances of Backward Simulation with 5000 Samples at Different Kurtosis Values

The data for samples of size 5000 are presented in Figure 5.3. Since collecting a complete dataset for 5000 samples could have taken considerable time, I chose to concentrate on values that appeared important in the 1000 sample case. Parallels exist

between the results. Again I observed that all peaked runs appeared to have their minima at lower quantized CPU speeds than Kurtosis 1. And again the Kurtosis 0.125 had a minimum considerably higher than the minimum of Kurtosis 1. Unlike the 1000 sample runs none of the different kurtosis values tried yielded statistically significant performance improvements in an absolute sense, though again it is worth noting that the leftward shift could constitute a performance improvement.

I also collected small quantities of data for sample sizes of 20000. Each run of this set could take up to two weeks to complete, so the data are necessarily sparse. The results are summarized in Figure 5.4.

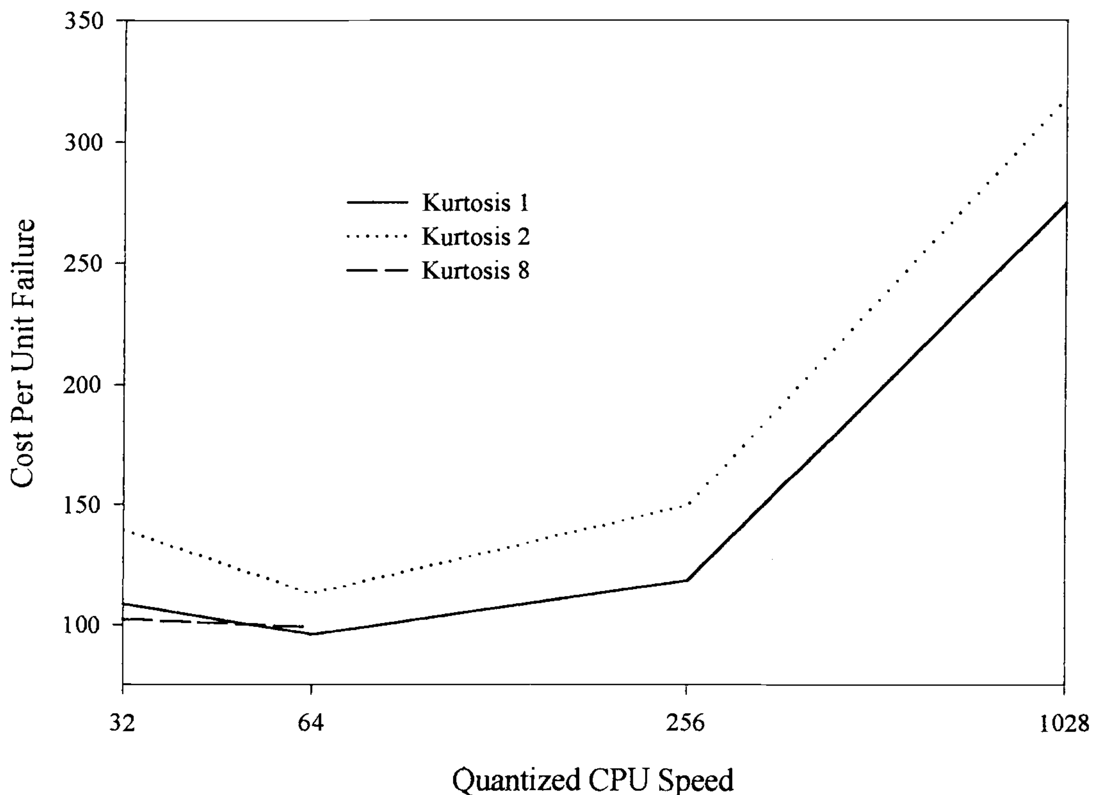

Figure 5.4: Average Performances of Backward Simulation with 20000 Samples at Different Kurtosis Values

Since the number of data points presented in Figure 5.4 is necessarily small, I can only state that the data suggests comparable performance between Kurtosis 8 and Kurtosis 1 runs, and that Kurtosis 2 appears to induce inferior performance.

Appendix B also includes a small portion of data for sample sizes of 50 and 100, but I concluded that these samples sizes produced insufficient performance to be seriously considered, regardless of the quantized CPU speed. Consequently, I did not complete a data set and will not report them here. However, they are collected in the appendix for the curious reader.

Even with the care taken to collect the data seen here, one cannot draw broad conclusions. It appears that this form of kurtosis does not significantly improve performance, though under tight time constraints a modest performance improvement of no more than 10% may be expected. Some of the improvement results from the leftward shift of the characteristic curve for the algorithm when kurtosis is applied. Also, it appears that while kurtosis may contribute some improvements, it must be tuned for maximal effect. Too much or too little kurtosis may degrade performance.

The kurtosis implemented in the Backward Simulation algorithm may not have the desired focusing effect. Instead of yielding a more accurate sample when sample size is limited, it may merely reduce the variance in such a sample. This may be desirable when consistency among choices is needed, but is not generally the same as concentrating computational effort on gathering high probability mass elements in a conditional distribution. Hence a more carefully designed "peaking" algorithm may demonstrate better performance with a focusing mechanism.

The methodology developed here is a useful contribution. By analyzing the properties of algorithms and associating them with domain characteristics, the possibility of designing better algorithms is within reach.

Clearly a better model of the OLMA is necessary to obtain a better understanding of what may be attributed to the algorithms and what are epiphenomena of processes within the OLMA. I will now turn my attention to this.

6. Analysis of the Random Algorithm

6.1 Introduction

Design and analysis of decision algorithms is a central aim of this dissertation. In order to effect this, it is necessary to understand the conditions under which algorithms will perform well. Complex testbeds are a necessary testing and comparison component for anytime algorithms, but they introduce a number of problems for the empirical scientist:

- Testbeds introduce additional variables that must be accounted for in algorithmic comparisons.
- Questions of optimality and ceiling or floor effects cannot usually be answered by empirical data alone, but must be accounted for by design and analysis of the testbed.
- Proper performance of different algorithms should be verified using all reasonably available methods.
- Attributing trends in observed data to the algorithm (as opposed to the testbed) requires knowledge of the testbed's characteristics

An accurate theoretical model can help answer these questions by providing the means to perform a more abstract analysis than that afforded by the data.

This chapter introduces two models of the OLMA testbed with the agent using the random action algorithm. The first was derived from a basic analysis of the testbed's structure, and will be shown to be inadequate for prediction of the testbed's behavior under different inputs. A stochastic automata network (SAN) model is then developed

which captures some of the behaviors missed by the simpler model. The SAN model developed is necessarily a discrete-time Markov chain model, and will draw on the development of the continuous-time SAN models of chapter 2. Lastly, I draw some conclusions about the capabilities of the models and suggest some improvements to testbed design that should aid future researchers.

6.2 Derivation of the Basic Cost Equations for the Random Algorithm

Recall that the random algorithm wastes a specific amount of time (namely, it calculates a specific number of stochastic samples, then throws the information gathered away). After wasting time, it selects one of the seven possible actions at random, and the agent executes this. What is the behavior of this algorithm? What should the long-term average cost per unit failure look like as a function of the quantized CPU time?

The On-Line Maintenance Agent has three primary parameters which control the amount of computation performed by the agent: the sim-cycles, decision time, and quantized CPU time. The sim-cycles parameter (*SC*) is the number of iterations of the testbed from the simulation's point of view, specifically the number of opportunities each gate has to break. Decision time (*DT*) is the estimated time for the agent to make a decision. It is an input parameter whose value is determined by empirical estimation before the execution of a run. When using the Random Algorithm, the decision time is a fixed value based on the number of iterations of stochastic simulation done before the agent takes its random action. A run's total length is the product of the decision time and the sim-cycles. The quantized CPU time (*Q*) may be thought of as the CPU speed of the processor used by the agent. The quantized CPU time divided by the decision time is the

ratio of time spent by the agent to the time spent by the simulator. Since the number of decision cycles is fixed, the simulator will execute fewer cycles relative to the agent with higher values of the quantized CPU time.

The probability of failure $P(F_{Simulator})$ is an input parameter, but it is fixed at 0.001 for all of the experiments in this dissertation. This is the probability of going into one of the three failure modes possible for each gate; hence the probability of a well gate remaining OKAY from one cycle to the next is 0.997. However, the input probability of failure is for use by the *simulator*, not the agent. One must make an adjustment for the relative speed of the agent to the simulator to calculate the probability of failure for an agent's decision cycle. The probability of failure for one failure mode of a component in the agent may be written as:

$$P(F_{Agent}) = \frac{\left[1-(1-(3*P(F_{Simulator})))^{\frac{DT}{Q}}\right]}{3} \qquad \textbf{Equation 6.1}$$

Both the agent and the simulator incur costs. The agent adds to the total cost whenever it performs a probe or a replacement action. It incurs no costs for doing nothing. In the half-adder model, there are two possible probes, four gates that may be replaced, and one "do nothing" action. The simulator adds to the total cost whenever it completes a cycle and the circuit is still broken. Cost is incurred for each gate broken per simulation cycle. The simulator incurs no costs if it is running correctly.

Consider the agent's costs alone as a function of the quantized CPU time during an extremely long (infinite, if you like) run. As the quantized CPU time approaches zero,

the ratio of time that the agent is spending relative to the simulation is decreasing. Thus if the agent is selecting an action at random, and incurring a consistent average cost over a long run of the testbed, the average cost from the agent should approach zero as the quantized CPU time approaches zero. That cost can be easily estimated since, on average, I expect two probes and four replacements out of every seven actions. Furthermore, if the decision time and sim-cycles are fixed, as they often are for a set of runs of the testbed, the agent's costs become a constant function of the quantized CPU speed. If I let C_P be the cost of a probe action, and C_R be the cost of a replacement, the agent's costs over the length of a run of the testbed may be approximated as:

$$C_{Agent} = \left[\frac{2C_P + 4C_R}{7} \right] * SC \qquad \textbf{Equation 6.2}$$

When we plot the agent's component as cost per unit failure, we must divide by the number of failures, which does vary with the quantized CPU speed. The agent's contribution to the cost per unit failure is then seen to be a positively sloped function. So the agent's cost per unit failure becomes:

$$CPUF_{AGENT} = \left[\frac{2C_P + 4C_R}{7 * P(F_{AGENT}) * 3} \right] \qquad \textbf{Equation 6.3}$$

Now consider the simulator's costs alone as a function of the quantized CPU time. The simulator will only incur costs when one of its gates is broken. This happens to any individual gate with failure probability $P(F_{Simulator})$. When a gate fails, I expect that on

the average it will take seven decision cycles to repair the gate, since the repair of a gate can be viewed as a Bernoulli variable. Failures are completely independent. As the quantized CPU time approaches zero, the simulator is stepping much faster than the agent is. Every gate will become broken in the simulator, and though the agent will sometimes fix it for a very small period of time, it will break again quickly from the agent's point of view. Thus the costs incurred from gates being broken will approach infinity as the quantized CPU time approaches zero.

As the quantized CPU time grows very large, the number of gates broken is very small. The broken gates are generally fixed after only one cycle of the simulator, so the costs should approach zero as the broken gate frequency drops. Thus the costs from the simulator approach zero as the quantized CPU time approaches infinity.

When a gate breaks, the agent on average will be halfway through a decision cycle. Then the agent will require seven more decision cycles to affect a repair. I can calculate how many total simulator cycles this is by using the decision time to quantum ratio and the number of decision cycles (*SC*). The simulator is charged 1 for each simulator time step that a gate is broken.

Now I may approximate the simulator's costs over a run of the testbed as:

$$C_{Simulator} = \frac{4 * 7.5 * SC * \frac{DT}{Q}}{\left[\frac{1}{P(F_{Simulator}) * 3} + 7.5 \right]} \qquad \textbf{Equation 6.4}$$

Equation 6.4 has an imprecision built into it: when the quantized CPU speed is very small, the time that transpires before a gate will spontaneously repair itself becomes comparable to the time needed for the agent to repair it. Augmenting Equation 6.4 to account for this is difficult. Note that here again the total cost must be divided by the expected number of failures to approximate the cost per unit failure contribution of the simulator.

Besides this obvious difficulty, one must ask whether the equations are sufficient to predict the performance of the testbed.

6.3 Analysis of the Cost Equations for the Random Algorithm

I may make some useful observations about Equations 6.3 and 6.4. How can the agent affect the quantities in these equations to obtain better performance? The agent cannot change the costs or the length of run. The agent cannot affect the probability of failure. In the first equation, the agent may affect the costs by choosing actions that are more appropriate to the situation: replacing the right gate when a replacement is needed, and doing nothing when the simulator appears to be okay. This is not necessarily as easy as it sounds. Several simulation cycles may transpire before evidence of the fault appears, and several decision cycles may be needed to accurately diagnose the problem. Ideally the agent will make a good guess quickly for its first repair attempt, then recover quickly and make a different second repair if that attempt fails.

In Equation 6.4, the agent can only affect the number of cycles used to make a correct repair. By making decisions quickly, the costs from this equation are minimized.

I can see how both equations are affected by altering the number of stochastic samples in the Random Action Algorithm. By taking more samples, the agent increases the decision time. Viewing the cost again as a function of quantum, this decreases the slope of the logarithmic equation 6.3. It also shifts the functional form *1/x* to the right in the simulator's equation.

So can the equations be used to predict performance? The graph of Figure 6.1 illustrates the total cost curve resulting from Equations 6.1 and 6.3. Also plotted are the sampled performance points for the algorithm for comparison.

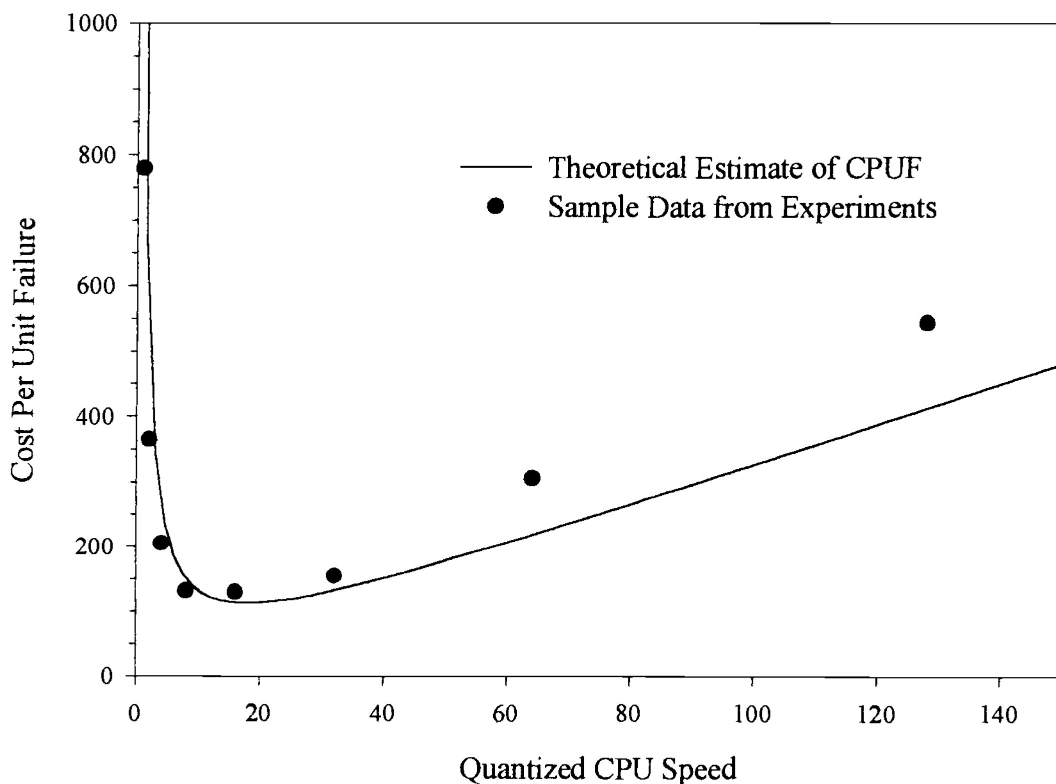

Figure 6.1: Comparison of the Random Action Algorithm with Sample Size 5000 Data with Theoretical Predictions

Though at first glance this appears a reasonable fit, I could not explain discrepancies between the theoretical model and the observed data. The discrepancies were slightly larger for the sample size 1000 data, and I was unable to reverse engineer the parameters from the observed data. Despite some allowances for the imprecision of the equations, something seemed amiss. This was particularly true for the rightmost portion of the graphs I was viewing: as with the graph of Figure 5.1, performance leveled off after a certain quantized CPU speed was exceeded. The particular quantized CPU speed seemed to depend on the sample sizes. Since the only thing affected by sample size was the decision time, it was logical to look more closely at this parameter.

What became apparent is that the testbed was programmed to reset the simulator if the agent had made a decision, rather than have the agent wait for the simulator to complete its cycle. The consequences become most apparent when the time taken by the simulator exceeds the time taken by the agent: at this point, the agent and simulator step together, and the ratio of decision time and quantized CPU speed in both equations becomes unity. Thus the total cost is approximately fixed whenever the simulator requires more time to complete a cycle than the agent does.

Approximately, but not quite. A decay process seemed to be lurking in the data, as once the quantized CPU speed exceeded the decision time the graph did not merely level off, but first peaked slightly higher, then showed a gradual decrease as if toward an asymptote. And I still was puzzled why data fitting in the lower quantized CPU speeds was so difficult.

Further analysis and data from a similar testbed [Krishnamurthy 1999] suggested that perhaps the resetting of the simulator was causing a stepping phenomenon to occur

for slower quantized CPU speeds. I had been collecting data at quantized CPU speeds that were powers of two. Comparing these values to ratios of quantized CPU speed and decision time revealed that if a stepping phenomenon existed, it could not show in the data I had collected. There were no two data points on the same step!

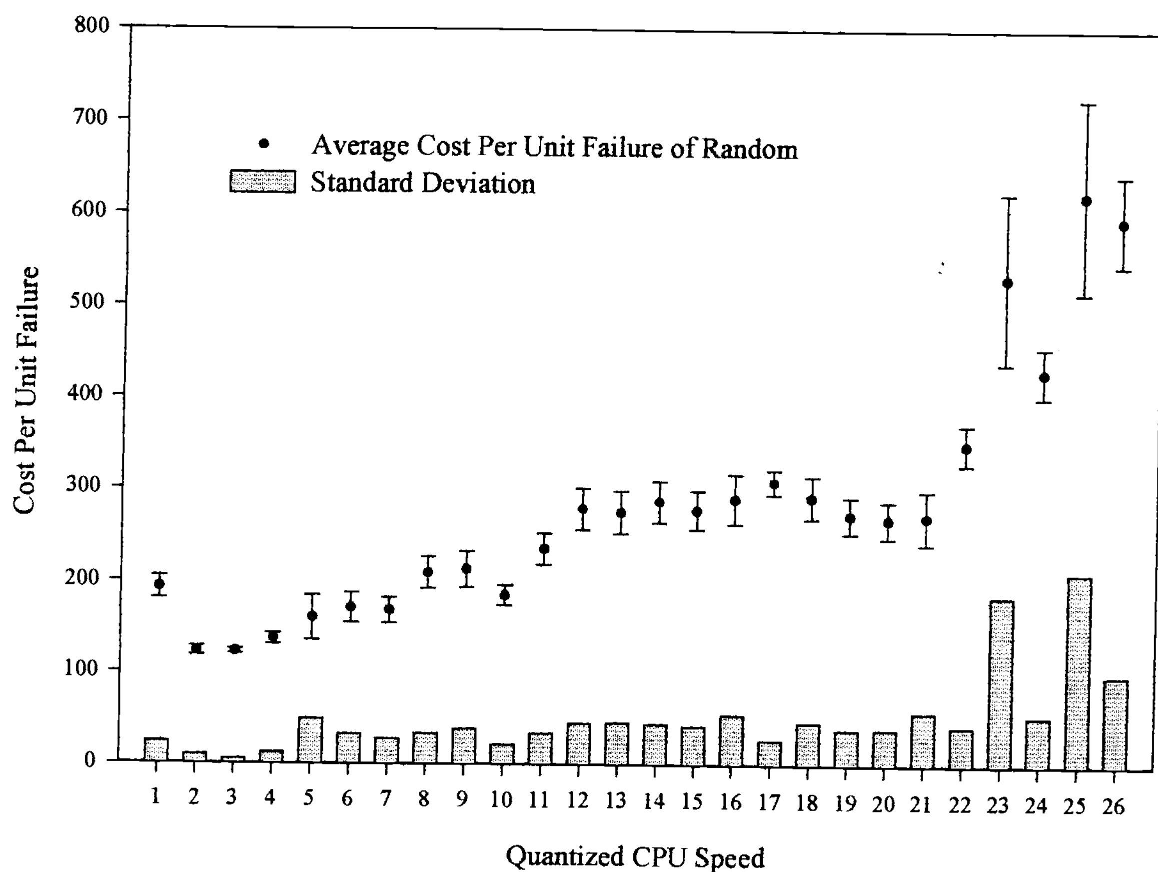

Figure 6.2: Cost Per Unit Failure of the Random Algorithm at 1000 Samples with 90% Confidence Intervals and Variances

Armed with this knowledge, I collected a complete set of data for quantized CPU speeds less than or equal to 26 for the Random algorithm using 1000 samples. The decision time for 1000 samples is 22.2 quantized units. I present the data in Figure 6.2. The error bars indicate a small sample confidence interval with n=8 and α=0.1, with the

appropriate t value 1.415 taken from Table A.4 of [Walpole et al. 1985]. The raw data and error values are available in Appendix C.

Note that the data appear to step at approximate fractions of 22.2, with a fairly clean step at quantized CPU speeds of 12 through 22. A step appears plausible at speeds 6 through 11, though the behavior there is not smooth and some epiphenomena may be creating irregularities. It is worth observing that there is absolutely no relationship between the quantized CPU speeds originally sampled (which were powers of two) and the quantized CPU speeds where these behavior shifts occur. Behavior above quantized CPU speed 23 is unexpectedly bizarre, with both costs and variances appearing to jump wildly from one speed to the next. And there may be a transition phenomenon occurring near the boundaries that are fractions of the decision time, as some local bumps appear in the graph at those points.

Of these, I can only suggest a plausible explanation for the lattermost: since the manually set decision time is only an approximate value, garbage collection and other internal LISP behaviors may occasionally cause the actual decision time required to exceed the expected value. This should happen most often at values immediately below fractions of the decision time (for example, 11 and 22) and should result in slightly higher costs at those points. While the data is suggestive of such behavior, this is merely a hypothesis at this time.

Still, the data suggests that this *synchronizing effect* within the testbed does exist, and that the equation models are inadequate to capture this behavior. How might we address this inadequacy? I found a candidate in Markov chain theory.

6.4 A Stochastic Automata Network Model

As noted in Chapter 2, Markov chain models provide a compact and easily analyzed representation of many stochastic systems. For our purposes, the most desirable property is the ability to calculate a steady-state probability distribution vector, which summarizes the long-term proportion of time the system will spend in various states. Since standard Markov models do not permit recombinant calculations of a complex automata network from its simpler constituent automata, stochastic automata networks have been developed to enable such calculations. Unfortunately, the mathematical machinery to perform the calculations is unnecessarily complicated for our simple domain.

In the OLMA, the agent repairs a component only if that component is broken. Otherwise the transition probabilities of the components remain unaffected. The agent's act of repairing a component is clearly a synchronizing transition since it forces a change of state in the Markov chain of the repaired component. Whether we also wish to consider this a functional transition may be left up to the modeler: one may view a repair action taken when the gate is okay as being a "do nothing." If we choose this view, then the state of the agent becomes a function of the state of the component(s) it repairs. For simplicity, I chose to model this as strictly a synchronizing transition and charge the agent for failed repairs.

I was faced with three possible ways of generating a SAN model for the OLMA: generate the entire 96 x 96 matrix by hand, use the general method for applying a sum of products to handle synchronizing transitions, or create a simple tensor product model augmented by local transformations for component repair. No generalized tensor product

applicable to agent repair scenarios has appeared in the literature recently [Stewart 99] and as of this writing, the models for handling synchronizing or functional transitions in discrete-time Markov chain models are significantly more complex than their continuous-time counterparts [Stewart 00]. However, I was able to calculate the global transition matrix of the OLMA by generating a tensor product model with transformations during the intermediate stages.

As we noted when discussing the equation models, the testbed inputs determine the relative time allocations between the agent and the simulator. This cannot be captured with a single SAN model, but I can describe it with a family of SAN models. When the agent and simulator are synchronizing on agent actions, the time ratio is one-to-one, and no adjustments are needed. But if the agent is running twice as fast, then I introduce an extra dummy state node in the Markov chain model of the agent. This must be accounted for in the final calculations of the costs of running the agent.

My model takes the view that component repair actions are a *probability transfer function* over the tensor product of the agent and the repairable components. Repair events transfer probability mass from one location to another within this matrix. Thus as long as I am modeling a single agent and independent components I can construct the tensor product with a record of the combination order and use nested loops to transfer probability mass from one location to another.

As an example, consider the case where the agent replaces Gate 1. The only time this repair action changes the state of Gate 1 is when the gate is broken. Thus I wish to transfer all probability mass from locations representing the agent transitioning to a state of Gate 1 Okay (regardless of the past state of the agent) when Gate 1 is broken and

transitioning to a state of remaining broken to corresponding locations in the same agent states but with Gate 1 broken and transitioning to a state of normal operation, with the values of all other gate states remaining unchanged. If I know the order in which the components and agent were combined, I can create an indexing function into the tensor product and transfer probability mass according to this strategy. Using a regular combination order with the last, leftmost tensor product combining the agent eases the calculation of the indices and subsequent error checking, as well as permitting the modeling of interacting components using the usual generalized tensor product (provided the interactions are independent of the agent).

Matlab code for representing and calculating the results for one submodel is given in Appendix D. Code for the several other models is similar. The different submodels capture discrete portions of the event space: the model constructed with Agent 1, for example, will represent the portion of the event space where the quantized CPU speed exceeds the decision time required by the agent, since then the agent and simulator will change state at the same rate. Agent 2 has an intermediate dummy state between decisions, and therefore the simulator is running up to twice as fast as the agent, and so forth. Note that finding a tensor matrix to represent the global Markov chain is only the first step, as I need to know how much time the system spends in each compound state over the long term. The equilibrium probabilities are found by calculating the appropriate 1-eigenvectors for each tensor product matrix. Once I had these, I calculated the costs for running the system: the agent costs resulting from the agent replacing or probing, and simulator costs incurred when a component is broken. Much of this was done with a hand calculator, since it enabled crosschecking of intermediate results.

The first several figures illustrate the graphical models of the agent used to generate the stochastic automata networks (SANs) described in Chapter 6. Since the simulator can run significantly faster than the agent, a SAN model of the complete system necessarily will comprise several submodels. Since each gate component is identical to every other gate component, we can model each of these gates as a simple stochastic automaton. When the agent and simulator are synchronized at a speed ratio of one to one, we can also represent the agent using a simple stochastic automaton. The transition matrices for each may be combined directly as a tensor product with only minor adjustments to account for agent actions that change the state of simulator components. When the simulator runs twice as fast as the agent, the model of the agent requires an extra node to stall its speed to match that of the simulator. When the simulator is three times as fast, the agent will have two extra nodes. And so on.

Matlab code for calculating the global transition matrices follows the graphical models. The code will be clear if the reader keeps in mind that changes are only made to entries representing the agent replacing broken components.

6.5. Models for Agent and Simulator Speed Ratios

Choosing a convenient representation can be something of an art. For modeling the behavior of the agent and simulator using the random algorithm, I chose not to represent the possible states of the system because of the size of the resultant representation. I could easily estimate the costs the system would incur from the simulator's state changes: the simulator will induce costs for each time cycle that a gate is broken, and the simulator is responsible for changing a gate from the OKAY state to

BROKEN according to fixed probabilities. What remained uncertain was how long the agent would take to repair the circuit, and what costs might be incurred in the interim. The difficulty is compounded when one realizes the agent's propensity to reset the simulator after choosing an action means that the agent is restricted to speeds that are multiples of the simulator's speed. Consequently, the models presented below focus on the decision process of the agent and its relationship to the simulator.

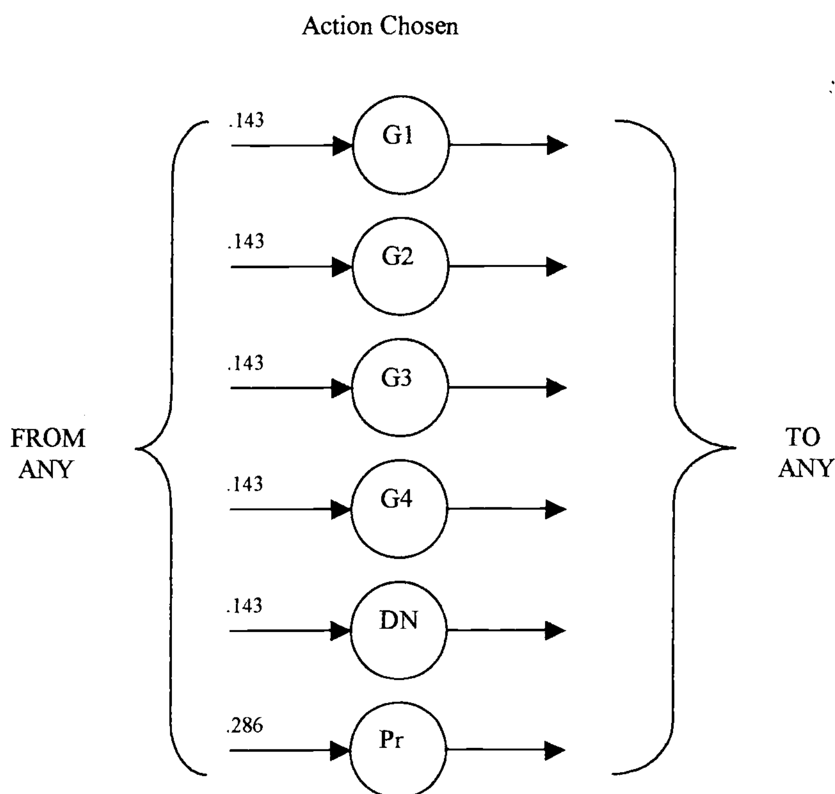

Figure 6.3: Graphical Representation of the Decision Process When Agent and Simulator Run at the Same Speed Using the Random Algorithm

In Figure 6.3 we show the graphical model of the decision process when the agent and simulator run at the same pace. Nodes G1 through G4 represent the agent choosing a

repair action on the appropriate gate. Nodes labeled DN and Pr represent Do Nothing and Probe, respectively. The agent will transfer from one state to another according to the probabilities labeled on the incoming arcs. Since the agent will reset the simulator upon choosing an action, this is also the model for any situation where the agent can make decisions in less time than the simulator needs to automatically change state.

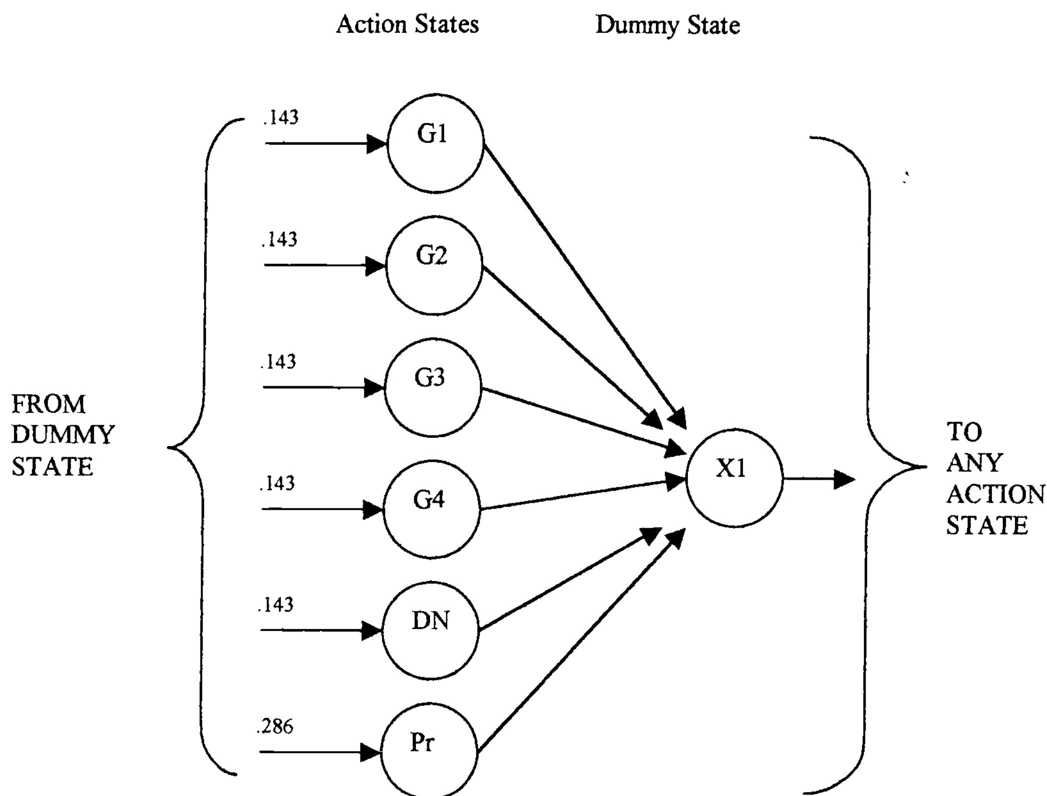

Figure 6.4: Graphical Representation of the Decision Process When Simulator Speed is Twice Agent Speed, With the Agent Using the Random Algorithm

In Figure 6.4 we illustrate the model for the case of the simulator making changes to component states twice as often as the agent chooses an action. The dummy state X1 represents the time during which the agent is not taking an action—hence it results in no

state changes in the combined model. The transition from an action selection state to the dummy state must occur at the next time step, hence there is no probability associated with this arc. Note again that since the agent resets the simulator upon choosing an action, this can occur for any speed setting where the simulator takes one action slightly prior to the agent making a decision and up to the point where simulator and agent are running at a ratio of 2:1.

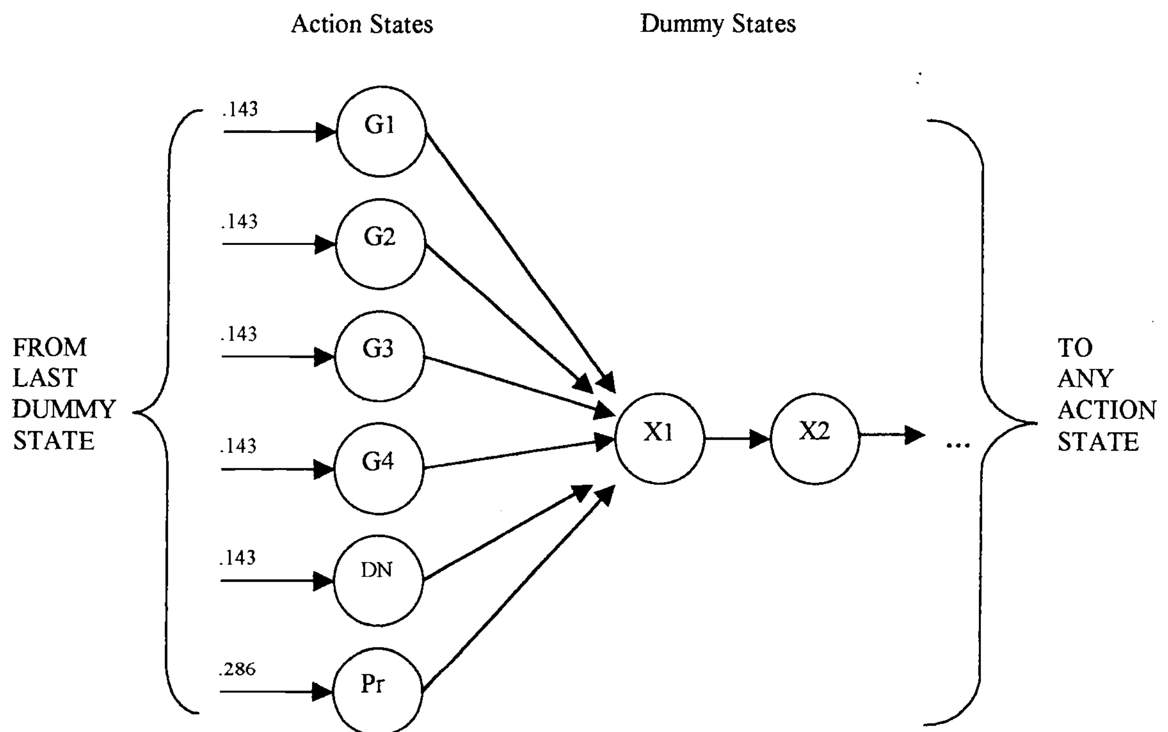

Figure 6.5: Generalizing of the Decision Process Model for Simulator/Agent Speed Ratios Greater than 1:1

We can generalize from this that the other agent models required simply have more than one dummy state. This is illustrated in Figure 6.5, where the states X1, X2... exist merely for marking time.

The gates are somewhat simpler to model, as they transition from OKAY to STUCK-1, STUCK-0, or UNKNOWN stochastically. We leave it to the reader to generate a graphical representation of the gates from the description in Chapter 3 if one is required.

6.6. Transition Matrices for Submodels of the Stochastic Automata Network

Having created stochastic automata to represent the agent when the agent's speed doesn't match the speed of the simulator, we can now discuss the needed transition matrices, and the semantic interpretation of combining them.

$$GATE = \begin{bmatrix} 0.997 & 0.003 \\ 0.003 & 0.997 \end{bmatrix}$$ **Equation 6.5**

In Equation 6.5 we see the 2 x 2 transition matrix that represents the normal behavior of a gate element in the simulator. The rows represent states prior to a time step, while the columns represent the state of the system after that time step. The numbers in the (row,column) positions represent the probability of arriving in the state represented by the column given that the gate is in the state represented by the row in the previous time slice.

As stated earlier, one can create a SAN for several gates by computing the tensor product of the matrices representing the individual gates. In my testbed, all of the gates exhibit the same behavior as the matrix of Equation 6.5 represents. Hence the tensor product is symmetric. Note that each row represents one of the n^x possible joint states of the n gate system, where each gate has x possible states, and that the (row,column)

positions now represent the probability of transitioning from the joint state represented by the row to that represented by the column.

$$AGENT1 = \begin{bmatrix} 0.143 & 0.143 & 0.143 & 0.143 & 0.143 & 0.286 \\ 0.143 & 0.143 & 0.143 & 0.143 & 0.143 & 0.286 \\ 0.143 & 0.143 & 0.143 & 0.143 & 0.143 & 0.286 \\ 0.143 & 0.143 & 0.143 & 0.143 & 0.143 & 0.286 \\ 0.143 & 0.143 & 0.143 & 0.143 & 0.143 & 0.286 \\ 0.143 & 0.143 & 0.143 & 0.143 & 0.143 & 0.286 \end{bmatrix}$$ **Equation 6.6**

The transition matrix for the agent has a slightly different interpretation. The (row,column) positions of the agent when the agent and simulator are running at the same speed are given in Equation 6.6. Each position represents the probability of the agent transitioning to a state of having chosen the action represented by the column when previously the agent had been in the state of choosing the action represented by the row. In this instance, all of the rows are identical because the choice of a new action is not conditioned on past actions (doing nothing is twice as probable as any other action). The action chosen can alter gate states in the joint SAN model, so it will be important to track action states. Note that I've modeled the state as persistent, while the effect of choosing an action in the testbed is almost instantaneous.

$$AGENT2 = \begin{bmatrix} 0 & 0.143 & 0.143 & 0.143 & 0.143 & 0.143 & 0.286 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$ **Equation 6.7**

In Equation 6.7 we can see the effect of adding a dummy state to the agent to resolve timing issues. The dummy node from Figure 6.4 now represents a state of having chosen a non-action. This non-action will not incur costs in the testbed, nor will it change gate states; thus it is similar to choosing to do nothing. However, the probability of going into this non-action state given an action was chosen (which includes doing nothing) is exactly 1. We can extend the timing correction for the necessary integer multiples of the simulation time by adding additional dummy nodes and adjusting the agent matrix. The matrix for the agent with two dummy nodes is given in Equation 6.8.

$$AGENT3 = \begin{bmatrix} 0 & 0.143 & 0.143 & 0.143 & 0.143 & 0.143 & 0.286 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \textbf{Equation 6.8}$$

6.7 Evaluation of the SAN model

Figure 6.6 compares the results of the SAN model described above with the detailed sample averages of the agent. Note that the SAN model more closely approximates the step behavior of the testbed. Some behavior near the decision time fractional transition points is unexplained by this model, but may be due to some feedback within the testbed resulting from the use of streams to communicate data between the agent and the simulator, or perhaps too from garbage collection or time-

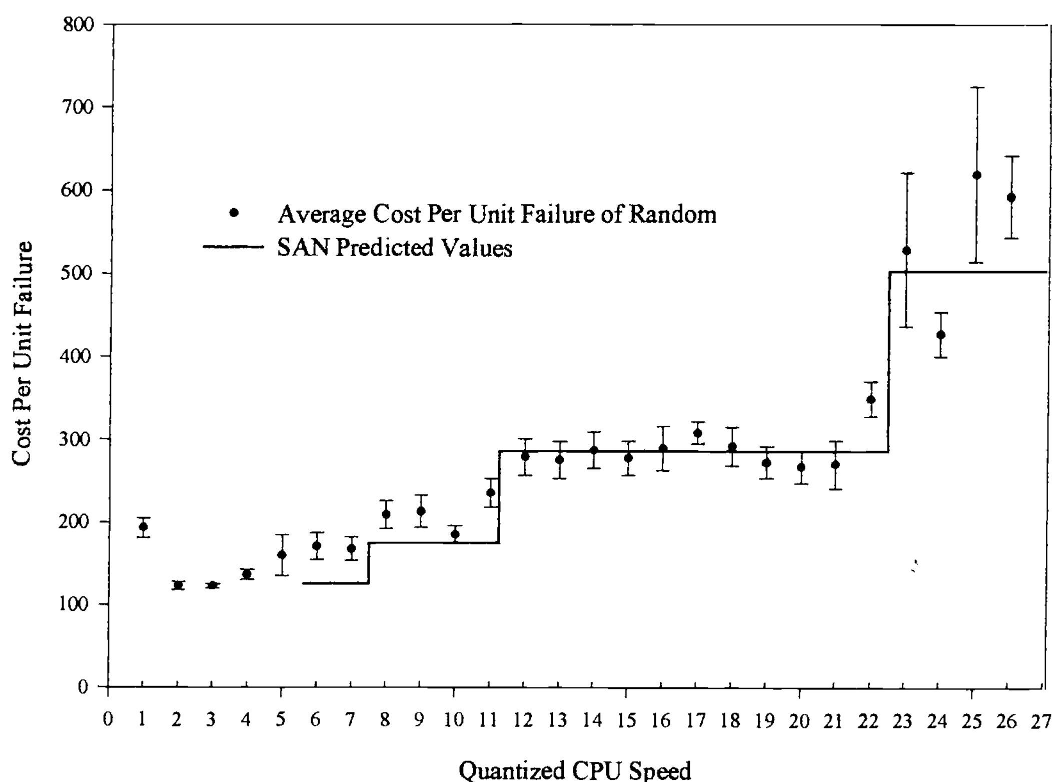

Figure 6.6: Comparison Between Stochastic Automata Network Predictions of Random Performance and Actual Random Performance

sharing throwing processor times off. Error bars represent a 90% confidence interval for the mean.

The equation and SAN models allow some additional comparisons, too. For one, imagine a perfect rational agent, one that always makes a correct repair whenever a component is broken. We will assume that the fault manifests itself during all future simulation cycles and that the agent needs no deliberation time before taking the proper corrective action. Note that some costs will still be incurred: though the agent will never probe, it still must replace some components. And a component may remain broken for several simulator cycles if the agent has to repair other components or if the simulator is

cycling much faster than the agent. Still, under most circumstances, when a component breaks during one cycle, it is immediately repaired during the next agent cycle. The cost of the component being broken one cycle is 1, and the cost of repair is 10. Thus the cost per unit failure when the simulator and agent are running at the same speeds is a tiny fraction over 11, since our perfect agent will detect the broken gate after one cycle and make the repair immediately in most cases.

This jumps to a little over 11.5 when the simulator is twice as fast as the agent, since we will still do one repair for the fault (at a cost of 10), but the gate will be broken for 1.5 cycles on average (one cycle half of the time, two cycles the other half). The cost jumps to 12 when the simulator is three times as fast as the agent, and so on. The optimal values will continue to jump in steps due to partial synchronization of the agent and simulator. As the quantized CPU speed approaches zero, this cost per unit failure will continue growing without bound; as quantized CPU speed grows toward infinity, the CPUF approaches the value 11 asymptotically.

This analysis depends largely on the equation model of performance and due to its assumptions is too conservative. I can generate a more liberal model of a perfect rational agent using the SAN model. The SAN model can tell me approximately how often gates will break, say once every 28.7 cycles. If one cycle is needed for detection of the fault (costing 1) and replacement cost is 10, then my average total cost will be approximately 383.2 over 1000 cycles. Dividing by the expected number of faults yields the expected cost per unit failure.

Since this construction attempts to treat gates individually and does not adjust for instances where multiple gates are broken, it should be slightly more costly than the

"true" minimum. Thus the two models together can provide us with a region where the lower bound on performance should occur. This is illustrated with several anytime algorithms in Figure 6.7.

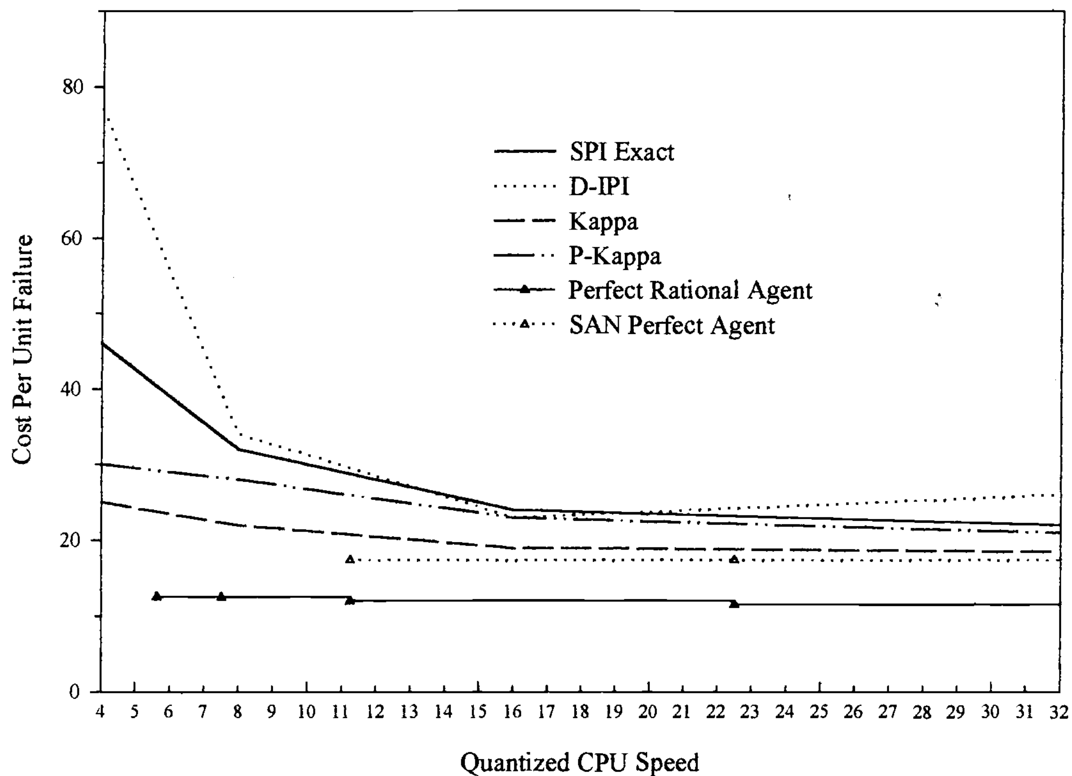

Figure 6.7: Comparison of Two Perfect Agent Models With Performance of Several Anytime Inference Models

I may now make an exciting observation: the Kappa and Posterior Kappa Reduced algorithms perform remarkably near optimal on the OLMA testbed! The performance of these algorithms is even more remarkable when one considers that fault detection contributes to the cost of the anytime algorithms.

One may also ask if I can approximate the behavior of our best algorithm, the Posterior Kappa Reduced, by estimating the probability the agent will repair a broken

component. Such an estimate is complicated by the inclusion of probe actions, since they are separate from the replacement actions and they also incur costs. Theoretically, one could generate a solution surface illustrating possible tradeoffs in the respective probabilities of doing nothing, replacing a component, or probing. This exercise does not appear to yield useful results, since I already know that the Posterior Kappa Reduced algorithm is close to optimal throughout its range. Then probability mass and resultant costs are shared primarily between replacement actions and doing nothing.

6.8 Conclusions

My efforts to model the behavior of the testbed under different algorithms presented some unusual challenges but yielded useful results. Chief among the difficulties is capturing the behavior of more complex algorithms. No easy solutions are forthcoming, though I would like to move the design and analysis of reasoning algorithms in this direction!

While modeling the Random algorithm proved to be more complex than initially anticipated, I uncovered information useful in the current line of research as well as a practical model for single agent repair scenarios. The latter is, of course, the application of SAN theory to the OLMA. My outline of the approach should be adequate for those familiar with the basics of SANs.

The implications for my algorithmic research are more immediate:

- Since the Posterior Kappa Reduced algorithm performs near optimal on the half-adder problem, we have uncovered a potential ceiling effect [Cohen 95] for future experiments. More difficult problems need to be coded into the testbed to test the

limits of this algorithm and to compare it to other algorithms with comparable performance.

- The synchronizing effect in the testbed causes anomalous behavior and should be eliminated.
- Some odd behaviors in the testbed may result from the current method of discretizing time. The use of a continuous time model borrowing ideas from work in simulation should be considered. In particular, I am interested in allowing continuous, variable transition rates managed by a central event queue.
- Testbeds for comparing reasoning algorithms should be designed in combination with a theoretical model to aid in experimental design and evaluation.

7. Conclusions

7.1. Summary of Findings

I have presented a number of empirical and analytic analyses in the three previous chapters. Several broad conclusions present themselves:

- The OLMA testbed confirms that reasonable testbeds can be constructed and used for analyzing reasoning algorithms.
- Several anytime algorithms demonstrate a time-to-quality tradeoff: that is, an agent may effectively use more computation time to improve results, and this can be done in a relatively continuous fashion.
- Some anytime algorithms appear to perform better than others, and this performance difference is independent of properties attributable to the testbed or its interaction with the agent.
- While our attempts to quantify kurtosis are an apparent failure, they suggest that variance plays a minor role in the effectiveness of the anytime algorithms.
- Equation models of performance may be easy to generate but ineffective for extrapolation. In particular, the testbed must be constructed carefully to avoid complex epiphenomena.
- Stochastic Automata Networks provide a strong analytic tool that can be used in conjunction with testbeds to clearly attribute observed behaviors with the testbed or the algorithms being analyzed. By taking the view that interactions between components were a probability transfer function, I was able to construct a SAN for the OLMA running the Random algorithm. In our testbed, we have not only

explained previously inexplicable structure in the observed algorithm data, but also shown that an optimal agent does not perform significantly better than the best of the anytime algorithms.

- Theoretical models may not explain all observed phenomena in complex testbeds. Operating system process scheduling policies, for example, may interact with the testbed in unanticipated ways. The best defense against the potential criticism that your observed data is not a consequence of what you measured but how you measured it is fully utilizing the strong theoretical models wherever possible.
- The hidden results are those of methodology. I have taken some care in presenting details of my data collection and analysis because it took me years to realize the benefits of that rigor. You, dear reader, should not have to repeat my follies. Further details concerning the data are presented in the appendices that follow. If you are fortunate enough to read this: go thou and do likewise—or better!

7.2. Future Research

Several avenues of research remain open to inquiry. I mention a few possibilities here.

First, we must ask if the peakedness of sampling distributions contributes significantly to the performance of our anytime algorithms in this domain? While our attempts at quantifying kurtosis led us to other conclusions, this question remains open and deserves further research. This research could potentially be accomplished within the current testbed.

The current testbed does have some flaws, and one could address these flaws by developing a modified testbed. Analysis of the current testbed is inhibited by the deliberate discretization of its behavior. Creating a continuous-time testbed using an event scheduler would allow the use of a continuous SAN model. Since calculations of the continuous joint model are easier in the continuous case, larger circuits could be analyzed with ease.

I have developed one such larger circuit that would truly test the capabilities of the OLMA: a nine-gate master-slave flip-flop. This circuit was originally intended to be part of this dissertation, but other issues overshadowed its construction and evaluation. The circuit is not merely larger than the four-gate half adder used in this dissertation. The flip-flop also has a time delay between when inputs arrive and when outputs are produced. Hence to make observations an agent should be capable of reasoning from past events to future consequences. I believe this can be done by a simple extension of the current dynamic Bayesian network implemented by the OLMA, and I hope some future student will consider carrying this work forward.

Bibliography

[Astrom 65] Astrom, K. "Optimal Control of Markov Processes with Incomplete State Information," in *Journal of Mathematical Analysis and Applications*, **10** (1965): 174-205.

[Bayer 99] Bayer, Valentina. "Approximation Algorithms for Solving Cost Observable Markov Decision Processes." Technical Report 99-50-01, Department of Computer Science, Oregon State University, 1999.

[Bouckaert 94] Bouckaert, Remco R. "A Stratified Scheme for Inference in Bayesian Belief Networks," in *Uncertainty in Artificial Intelligence: Proceedings of the Tenth Conference.* San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1994, pp 110-117.

[Cassandra 94] Cassandra, A. "Acting Optimally in Partially Observable Stochastic Domains." Brown University Technical Report CS-94-20, Department of Computer Science, Brown University, 1994.

[Cassandra 97] Cassandra, A., M. Littman and N. Zhang. "Incremental pruning: A simple, fast, exact method for partially observable Markov decision processes," in *Uncertainty in Artificial Intelligence: Proceedings of the Thirteenth Conference.* San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1997, pp 54-61.

[Charniak 91] Charniak, Eugene. "Bayesian Networks Without Tears," in *AI Magazine* **12** (4) (Winter 1991): 50-63.

[Cohen 95] Cohen, Paul R. *Empirical Methods for Artificial Intelligence.* Cambridge, Massachusetts: The MIT Press, 1995.

[Cooper 88] Cooper, Gregory F. "A Method for Using Belief Networks as Influence Diagrams," presented at the 1988 Workshop on Uncertainty in Artificial Intelligence. Minneapolis, Minnesota, 19-21 August 1988.

[Cooper 90] Cooper, Gregory F. "The Computational Complexity of Probabilistic Inference Using Bayesian Belief Networks," in *Artificial Intelligence* **42** (2-3): 393-406.

[D'Ambrosio 92] D'Ambrosio, Bruce. "Real-time Value-driven Diagnosis," in *Proceedings of the 3rd International Workshop on the Principles of Diagnosis*, 1992.

[D'Ambrosio 93] D'Ambrosio, Bruce. "Incremental Probabilistic Inference," in *Uncertainty in Artificial Intelligence: Proceedings of the Ninth Conference.* San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1993, pp 301-308.

[D'Ambrosio 96] D'Ambrosio, Bruce and Scott Burgess. "Some Experiments with Real-time Decision Algorithms," in *Uncertainty in Artificial Intelligence: Proceedings of the Twelfth Conference.* San Francisco: Morgan Kaufmann Publishers, 1996, pp 194-202.

[D'Ambrosio 96b] D'Ambrosio, Bruce. Personal communication.

[D'Ambrosio 99] D'Ambrosio, Bruce. "Inference in Bayesian Networks," in *AI Magazine* **20** (No. 2) (Summer 1999): 21-36.

[Draper 94] Draper, Denise L. and Steve Hanks. "Localized Partial Evaluation of Belief Networks," in *Uncertainty in Artificial Intelligence: Proceedings of the Tenth Conference.* San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1994, pp 170-177.

[Draper 95] Draper, Denise Lynn. *Localized Partial Evaluation of Belief Networks.* Ph.D. dissertation, University of Washington, 1995.

[Druzdzel 94] Druzdzel, Marek J. "Some Properties of Joint Probability Distributions," in *Uncertainty in Artificial Intelligence: Proceedings of the Tenth Conference.* San Francisco: Morgan Kaufmann Publishers, 1994, pp 187-194.

[Fernandez 96] Fernandez, Paulo; Brigitte Plateau; and William J. Stewart. *Numerical Issues for Stochastic Automata Networks.* Montbonnot St Martin, France: Institut National de Recherché en Informatique et en Automatique (INRIA Rhône-Alpes), Rapport de recherché N° 2938, 16 Juillet 1996.

[Filar et al. 1997] Filar, Jerzy and Koos Vrieze. *Competitive Markov Decision Processes.* New York: Springer-Verlag New York Inc., 1997.

[Fung 94] Fung, Robert and Brendan Del Favaro. "Backward Simulation in Bayesian Networks," in *Uncertainty in Artificial Intelligence: Proceedings of the Tenth Conference.* San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1994, pp 227-234.

[Geman 84] Geman, Stuart and Donald Geman. "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images," in *IEEE Transactions of Pattern Analysis and Machine Intelligence* **6** (No. 6) (November, 1984): 721-741.

[Goldszmidt 95] Goldszmidt, Moises. "Fast Belief Updating Using Order of Magnitude Probabilities," in *Uncertainty in Artificial Intelligence: Proceedings of the Eleventh Conference.* San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1995, pp 208-216.

[Heckerman 95] Heckerman, David and Michael P. Wellman. "Bayesian Networks," in *Communications of the ACM* **38** (3) (March 1995): 27-30.

[Horvitz 88] Horvitz, Eric J. "Reasoning Under Varying and Uncertain Resource Constraints," in *Proceedings AAAI-88: Seventh National Conference on Artificial Intelligence*, pp 111-116.

[Howard 60] Howard, R. A. *Dynamic Programming and Markov Processes.* Cambridge, MA: The MIT Press, 1960.

[Jaakola 94] Jaakola, Tommi, Michael I. Jordan, and Satinder P. Singh. "Reinforcement Learning Algorithms for Partially Observable Markov Decision Problems," in *Advances in Neural Information Processing Systems 7*, edited by Gerald Tesauro, David Touretzky, and Todd Leen. Cambridge, MA: The MIT Press, 1995, pp 346-352.

[Jensen 96] Jensen, Finn V. *An Introduction to Bayesian Networks.* New York: Springer-Verlag New York, Inc., 1996.

[Kaul 91] Kaul, Lothar. *A Feasibility Study on Compiling Reactive Problem Solution Methods for an AI Domain.* Master of Science Thesis, Department of Computer Science, Oregon State University, 1991.

[Krishnamurthy 99] Krishnamurthy, Ravi. Personal communication and portions of an unpublished project submitted for the Master of Science degree at Oregon State University, 1999.

[Li and D'Ambrosio 94] Li, Zhao-yu, and Bruce D'Ambrosio. "Efficient Inference in Bayes Nets as a Combinatorial Optimization Problem," in *International Journal of Approximate Reasoning* **11** (1) (1994): 55-81.

[Littman 95] Littman, Michael L., Thomas L. Dean, and Leslie Pack Kaelbling. "On the Complexity of Solving Markov Decision Problems," in *Uncertainty in Artificial Intelligence: Proceedings of the Eleventh Conference.* San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1995, pp 394-402.

[Littman 95b] Littman, Michael L., Anthony R. Cassandra, and Leslie Pack Kaelbling. "Learning Policies for Partially Observable Environments: Scaling Up," in *Machine Learning: Proceedings of the Twelfth International Conference on Machine Learning (ICML-95).* San Francisco, CA: Morgan Kaufmann Publishers, Inc., 1995, pp 362-370.

[Lusena 98] Lusena, Christopher, Judy Goldsmith, and Martin Mundhenk. "Nonapproximability Results for Markov Decision Processes." Technical Report TR-275-98, Department of Computer Science, University of Kentucky, 1998.

[Melekopoglou 94] Melekopoglou, Mary, and Anne Condon. "On the Complexity of the Policy Improvement Algorithm for Markov Decision Processes," in *ORSA Journal on Computing*, **6** (2) (Spring 1994): 188-192.

[Mundhenk 00] Mundhenk, Martin, Judy Goldsmith, Christopher Lusena, and Eric Allender. "Complexity of Finite-Horizon Markov Decision Process Problems," in *Journal of the Association for Computing Machinery*, to appear.

[Neapolitan 90] Neapolitan, Richard E. *Probabilistic Reasoning in Expert Systems: Theory and Algorithms.* New York: John Wiley & Sons, Inc., 1990.

[Papadimitriou 87] Papadimitriou, Christos H. and John N. Tsitsiklis. "The Complexity of Markov Decision Processes," in *Mathematics of Operations Research* **12** (3) (August 1987): 441-450.

[Parr 95] Parr, Ronald, and Stuart Russell. "Approximating Optimal Policies for Partially Observable Stochastic Domains," in *IJCAI-95: Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, Volume 2.* San Mateo, CA: Morgan Kaufmann Publishers, 1995, pp 1088-1094.

[Pearl 88] Pearl, Judea. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1988.

[Russell 95] Russell, Stuart and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1995.

[Schank 91] Shank, Roger C. "Where's the AI?" in *AI Magazine* **12** (4) (Winter 1991): 38-49.

[Smallwood 73] Smallwood, R. and E. Sondik. "The Optimal Control of Partially Observable Markov Processes over a Finite Horizon," in *Operations Research* **21** (1973): 1071-1088.

[Stewart 94] Stewart, William J. *Introduction to the Numerical Solution of Markov Chains.* Princeton, New Jersey: Princeton University Press, 1994.

[Stewart 95] Stewart, William J. "The Numerical Solution of Stochastic Automata Networks," in *The European Journal of Operations Research* **86** (3): 503-525.

[Stewart 99] Stewart, William J. Personal communication.

[Stewart 00] Stewart, William J. Personal communication.

[Tufte 83] Tufte, Edward R. *The Visual Display of Quantitative Information.* Cheshire, Connecticut: Graphics Press, 1983.

[Walpole et al. 1985] Walpole, Ronald E. and Raymond H. Myers. *Probability and Statistics for Engineers and Scientists.* New York: Macmillan Publishing Company, 1985.

[Westerberg 93] Westerberg, Caryl J. *Investigation of Automatic Construction of Reactive Controllers.* Master of Science Thesis, Department of Computer Science, Oregon State University, 1991.

APPENDICES

Appendix A: Cost-Per-Unit-Failure Performance Data for Several Anytime Inference Algorithms

A.1. Introduction

The data included here cover the experiments conducted in Chapter 4 and the UAI-96 paper with Bruce D'Ambrosio [D'Ambrosio and Burgess 1996]. Each section contains only the data averages rather than all of the raw data. All performance data reported here is *Cost Per Unit Failure.* Details of the experimental method can be found in Chapter 4. When D'Ambrosio generated a separate data set that was included in the UAI-96 paper, it is included here and labeled as such. As discussed in the text, the Random Algorithm data published then was grossly in error. In some data sets, not all of the points on a grid were collected; those not collected are marked as not available (N/A).

A.2. Data for the Kappa-reduced Algorithm

The data for the Kappa-reduced algorithm are averages of three runs per point reported for all of the data. The data sets here appeared to have similar values and variances.

		Quantized CPU Speed			
		1	**4**	**16**	**64**
Steps	**1**	**904**	**532**	**462**	**139**
	4	**550**	**190**	**141**	**254**
	16	**60**	**26**	**20**	**19**
	32	**51**	**25**	**19**	**18**
	64	**126**	**41**	**23**	**17**

Table A.1: Averages of Performance Data for the Kappa-reduced Algorithm.

Steps	Quantized CPU Speed				
		1	4	16	64
	1	850	380	500	135
	4	500	200	130	110
	16	68	28	20	19
	64	150	55	25	16.5

Table A.2: D'Ambrosio's Averages of Performance Data for the Kappa-reduced Algorithm as Published in UAI-96.

A.3. Data for the D-IPI Algorithm

My data are from one run of each data point, except for Quantum 1, Steps 1-64, which are averages of two runs each.

D'Ambrosio and I have been unable at this juncture to determine why there appear to be some discrepancies between the data in Tables A.3 and A.4, but some might be attributable to runs on different machines and compiler versions, and some may be due to sample variance. As discussed in Chapter 4, extremely short runs appear to have a "start up" effect which results in significantly lower cost per unit failure. Also, our paper in UAI-96 reports using a replacement cost of 3 per gate instead of 10, with an extent cost of 10 instead of 30, which may explain some differences in the data. Since

Steps	Quantized CPU Speed						
		1	4	8	16	32	64
	1	77	77	112	210	484	1223
	4	53	96	34	53	265	664
	16	101	83	37	23	80	102
	64	115	94	116	50	61	205
	256	850	131	51	38	26	N/A

Table A.3: Performance Data for the D-IPI Algorithm.

Steps		Quantized CPU Speed							
		1	2	4	8	16	32	64	128
	1	40.7	27.8	25.9	20.6	26.0	29.5	21.0	20.6
	2	37.3	34.4	28.0	27.3	28.7	21.0	26.0	22.0
	4	40.3	27.7	23.0	22.9	21.0	19.8	21.5	19.0
	8	N/A	41.2	35.9	25.4	23.3	30.0	19.2	35.0
	16	65.6	72.8	60.5	23.4	23.0	38.3	28.0	22.0
	32	59.2	63.6	48.0	24.8	21.9	18.0	23.0	29.0
	64	N/A	40.1	50.0	19.8	22.6	19.0	19.0	20.0
	128	N/A	107.0	63.0	32.0	23.5	15.0	18.6	18.8
	256	N/A	122.7	86.5	58.0	45.5	24.0	24.8	21.4

Table A.4: D'Ambrosio's Performance Data for the D-IPI Algorithm as Published in UAI-96.

D'Ambrosio's data collection method was not recorded and his data set is an amalgam of my data collection methods and his, this may explain the differences in the data sets.

A.4. Data for the Posterior Kappa-reduced Algorithm

Data averages in Table A.5 for quantized CPU speeds of 2 and 8 are based on one run, while the remainders are based on averages of two to four runs. Details of the method used to collect D'Ambrosio's data are not available, but it is reasonable to assume that these generally are averages of multiple runs and that the runs were probably

Steps		Quantized CPU Speed					
		1	2	4	8	16	64
	1	59	29	35	30	39	45
	4	70	28	30	28	25	19
	16	66	60	37	34	29	19
	64	95	69	54	62	23	20

Table A.5: Average Performance Data for the Posterior Kappa-reduced Algorithm.

mostly shorter 100 decision cycle runs, with some quantum values requiring larger decision cycle runs for reasons discussed in Chapter 4. Again it remains unclear whether some of D'Ambrosio's runs used a lower replacement cost, which would substantially improve performance.

		Quantized CPU Speed						
		2	4	8	16	32	64	128
Steps	1	41.4	33.1	23.9	21.8	28.4	24.3	21.0
	2	21.3	34.5	20.6	17.0	18.6	17.0	19.2
	4	25.0	26.0	20.0	38.9	20.0	18.9	18.6
	8	29.7	20.3	22.1	21.1	17.0	18.3	18.1
	16	36.9	23.0	18.5	17.0	17.7	20.6	15.9
	32	53.0	31.8	22.9	25.4	17.9	19.1	15.8
	64	76.3	41.9	29.9	31.6	17.3	15.0	18.6
	128	175.1	39.7	34.0	36.6	17.3	18.8	15.5
	256	90.7	43.0	29.0	40.2	14.8	18.0	14.1
	512	113.4	50.0	38.8	37.7	17.6	17.3	18.3

Table A.6: D'Ambrosio's Average Performance Data for the Posterior Kappa-reduced Algorithm as Published in UAI-96.

A.5. Data for the Exact Algorithm (SPI)

Averages reported here comprise three runs each for quantized CPU speeds of 1 and 4, and two runs each for quantized CPU speeds of 16 and 64. The data reported in UAI-96 for this algorithm are the minimum cost-per-unit-failure values, not the averages. As discussed in Chapter 4, the step size is not variable for this algorithm. I collected all of the data reported here, and this data was also the set used for our UAI-96 paper.

	Quantized CPU Speed			
	1	4	16	64
Step 1	145	46	24	20

Table A.7: Average Performance Data for the SPI Exact Algorithm.

	Quantized CPU Speed			
	1	4	16	64
Step 1	74	37	23	18

Table A.8: Minimum Cost-Per-Unit-Failure Values for the SPI Exact Algorithm as Published in UAI-96.

Appendix B: Raw Data for Experiments with the Peaked Backward Simulation Algorithm

B.1. Introduction

The data here represent runs with Backward Simulation with variable kurtosis, as described in the text of Chapter 5. All data runs in this appendix are organized in groups according to sample size, kurtosis value, and quantum. They were collected on a Sun Sparc Enterprise 4000 Server (locally known at Oregon State University as "Trek") running Sun Solaris. Each run required from four hours to two weeks of processor time to complete.

In very rare cases, runs produced extremely low numbers of failures for the length of run. It is not entirely unexpected that code constructed over a period of close to ten years, ported across three Lisp compilers and executed for another five years might have occasional difficulties. In these instances I chose to fault the code, toss out the resultant data point, and redo the run. All of these data points are documented, with the redo value immediately following the item it replaces.

Understanding the results: Each cluster of data is preceded by the input batch-run pattern for that cluster. The *numeric* inputs are the only ones relevant, and from left to right they are: quantized CPU value, steps (not relevant for the Random algorithm), probability of failure (for each failure mode of a gate in the simulator), decision time, sim-cycles (the approximate number of agent decisions made), kurtosis value, and number of samples.

The data lines are written as **cost / # failure = cost per unit failure**. Each cluster is averaged. This appendix includes a few extra data items not mentioned in the text due to the incompleteness of the data.

B.2. The Raw Data

(batch-run 1 100 .001 32 1000 t 1 1000)

62652 / 181	=	346.14		
60835 / 204	=	298.21		
59020 / 182	=	324.28		
63800 / 198	=	322.22		
			average =	322.71

(batch-run 2 100 .001 32 1000 t 1 1000)

23749 / 121	=	196.27		
25498 / 122	=	209.00		
19773 / 101	=	195.77		
23243 / 124	=	187.44		
			average =	197.12

(batch-run 4 100 .001 32 1000 t 1 1000)

10828 / 39	=	277.64		
10781 / 84	=	128.35		
11541 / 81	=	142.48		
9841 / 71	=	138.61		
			average =	171.77

(batch-run 8 100 .001 32 1000 t 1 1000)

5299 / 47	=	112.74		
4540 / 44	=	103.18		
4542 / 39	=	116.46		
5682 / 53	=	107.21		
			average =	109.90

(batch-run 16 100 .001 32 1000 t 1 1000)

3378 / 23	=	146.87		
3506 / 24	=	146.08		
3615 / 23	=	157.17		
3440 / 22	=	156.36		
			average =	151.62

(batch-run 32 100 .001 32 1000 t 1 1000)

2969 / 16	=	185.56		
2717 / 10	=	271.70		
2746 / 11	=	249.64		
2824 / 11	=	256.72		
			average =	240.91

(batch-run 64 100 .001 30 2000 t 1 1000)

5524 / 22	=	251.09		
5550 / 25	=	222.00		
5586 / 18	=	310.33		
5109 / 18	=	283.83		
			average =	266.81

(batch-run 128 100 .001 32 1000 t 1 1000)

2868 / 12	=	239.00		
2658 / 12	=	221.50		
2430 / 11	=	220.91		
2697 / 10	=	269.70		
			average =	237.78

(batch-run 256 100 .001 31 2000 t 1 1000)

5703 / 28	=	203.68		
5640 / 27	=	208.89		
5529 / 21	=	263.29		
5545 / 26	=	213.27		
			average =	222.28

(batch-run 1028 100 .001 31 8000 t 1 1000)

22147 / 92	=	240.73		
22004 / 109	=	201.87		
22057 / 88	=	250.65		
22044 / 90	=	244.93		
			average =	234.55

(batch-run 1 100 .001 32 1000 t 2 1000)

49934 / 249	=	200.54		
48868 / 250	=	195.47		
45030 / 231	=	194.94		
49884 / 250	=	199.54		
			average =	197.62

(batch-run 2 100 .001 32 1000 t 2 1000)

19523 / 131	=	149.03		
21489 / 151	=	142.31		
20542 / 148	=	138.80		
10089 / 50	=	201.78		
			average =	157.98

(batch-run 4 100 .001 32 1000 t 2 1000)

9908 / 80	=	123.85		
10522 / 91	=	115.63		
10178 / 86	=	118.35		
10197 / 88	=	115.88		
			average =	118.43

(batch-run 8 100 .001 32 1000 t 2 1000)

5747 / 40	=	143.68		
5667 / 43	=	131.79		
5621 / 43	=	130.72		
5595 / 35	=	159.86		
			average =	141.51

(batch-run 16 100 .001 32 1000 t 2 1000)

4743 / 24	=	197.63		
4440 / 23	=	193.04		
4788 / 24	=	199.50		
4605 / 20	=	230.25		
			average =	205.11

(batch-run 32 100 .001 32 1000 t 2 1000)

4352 / 15	=	290.13		
4271 / 13	=	328.54		
4103 / 12	=	341.92		
4279 / 13	=	329.15		
			average =	322.44

(batch-run 64 100 .001 31 2000 t 2 1000)

7797 / 14	=	556.93		
7450 / 15	=	496.67		
8252 / 26	=	317.38		
8256 / 26	=	317.54		
			average =	422.13

(batch-run 128 100 .001 32 1000 t 2 1000)

4171 / 12	=	347.58		
4088 / 12	=	340.67		
4103 / 12	=	341.92		
4513 / 8	=	564.13		
			average =	398.58

(batch-run 256 100 .001 31 2000 t 2 1000)

8175 / 22	=	371.59		
8164 / 27	=	302.37		
8587 / 26	=	330.27		
7530 / 17	=	442.94		
			average =	361.79

(batch-run 256 100 .001 32 1000 t 2 1000)

4166 / 12	=	347.17		
4304 / 14	=	307.43		
4293 / 12	=	357.75		
4533 / 16	=	283.31		
			average =	323.92

(batch-run 1028 100 .001 31 2000 t 2 1000)

6771 / 14	=	483.64		
7608 / 24	=	317.00		
7980 / 22	=	362.73		
8213 / 24	=	342.21		
			average =	376.40

(batch-run 1 100 .001 32 1000 t 8 1000)

44064 / 280	=	157.37		
42346 / 253	=	167.38		
43386 / 238	=	182.29		
37902 / 238	=	159.25		
			average =	166.57

(batch-run 2 100 .001 32 1000 t 8 1000)

16596 / 155	=	107.07		
17643 / 160	=	110.27		
16282 / 168	=	96.92		
16124 / 152	=	106.08		
			average =	105.08

(batch-run 4 100 .001 32 1000 t 8 1000)

8917 / 92 = 96.92
8630 / 102 = 84.61
8279 / 96 = 86.24
8471 / 91 = 93.09
average = 90.22

(batch-run 8 100 .001 32 1000 t 8 1000)

5719 / 42 = 136.17
5847 / 47 = 124.40
5765 / 46 = 125.33
5755 / 46 = 125.11
average = 127.75

(batch-run 16 100 .001 32 1000 t 8 1000)

4826 / 33 = 146.24
4896 / 17 = 288.00

(batch-run 16 100 .001 32 1000 t 8 1000)

4959 / 27 = 183.67
4837 / 23 = 210.30
4940 / 27 = 182.96
4971 / 23 = 216.13
average = 198.27

(batch-run 32 100 .001 32 1000 t 8 1000)

4840 / 10 = 484.00
4936 / 14 = 352.57
4787 / 12 = 398.92
4843 / 11 = 440.27
average = 418.94

(batch-run 64 100 .001 32 1000 t 8 1000)

4855 / 15 = 323.67
4635 / 10 = 463.50
4759 / 13 = 366.08
4807 / 15 = 320.47
average = 368.43

(batch-run 128 100 .001 32 1000 t 8 1000)

4807 / 11 = 437.00
4794 / 12 = 399.50
4904 / 15 = 326.93
4786 / 8 = 598.25
average = 440.42

(batch-run 256 100 .001 32 1000 t 8 1000)

4744 / 11	=	431.27	
4594 / 12	=	382.83	
4866 / 12	=	405.50	
4680 / 3	=	1560.00	
		average =	694.90

(batch-run 1 100 .001 32 1000 t .5 1000)

44364 / 247	=	179.61	
44110 / 233	=	189.31	
43456 / 241	=	180.32	
45750 / 237	=	193.04	
		average =	185.57

(batch-run 2 100 .001 32 1000 t .5 1000)

19138 / 161	=	118.87	
19432 / 161	=	120.70	
17743 / 144	=	123.22	
17264 / 158	=	109.27	
		average =	118.02

(batch-run 4 100 .001 32 1000 t .5 1000)

9718 / 93	=	104.49	
10892 / 104	=	104.73	
8999 / 70	=	128.56	
9278 / 79	=	117.44	
		average =	113.85

(batch-run 8 100 .001 32 1000 t .5 1000)

7333 / 63	=	116.40	
7146 / 51	=	140.12	
6906 / 46	=	150.13	
6613 / 46	=	143.76	
		average =	137.60

(batch-run 16 100 .001 32 1000 t .5 1000)

5260 / 33	=	159.39	
5336 / 28	=	190.57	
5242 / 24	=	218.42	
5220 / 27	=	193.33	
		average =	190.43

(batch-run 32 100 .001 32 1000 t .5 1000)

4809 / 17	=	282.88		
4598 / 20	=	229.90		
4705 / 21	=	224.05		
4746 / 14	=	339.00		
			average =	268.96

(batch-run 64 100 .001 31 2000 t .5 1000)

7649 / 25	=	305.96		
7943 / 11	=	722.09		
7940 / 25	=	317.60		
7711 / 24	=	321.29		
			average =	416.74

(batch-run 128 100 .001 32 1000 t .5 1000)

3617 / 13	=	278.23		
3569 / 8	=	446.13		
3800 / 12	=	316.67		
3906 / 8	=	488.25		
			average =	382.32

(batch-run 256 100 .001 32 1000 t .5 1000)

3311 / 13	=	254.69		
3196 / 11	=	290.55		
3105 / 12	=	258.75		
3321 / 12	=	276.75		
			average =	270.19

(batch-run 1 100 .001 32 1000 t .125 1000)

46043 / 223	=	206.47		
54796 / 245	=	223.66		
54629 / 213	=	256.47		
56615 / 226	=	250.51		
			average =	234.28

(batch-run 2 100 .001 32 1000 t .125 1000)

17061 / 102	=	167.26		
23817 / 140	=	170.12		
20415 / 141	=	144.79		
22493 / 136	=	165.39		
			average =	161.89

(batch-run 4 100 .001 32 1000 t .125 1000)

(toss)	6070 /	1	=	6070.00		
(redo)	9885 /	82	=	120.55		
	10777 /	79	=	136.42		
	11699 /	85	=	137.64		
	9914 /	77	=	128.75		
					average =	130.84

(batch-run 8 100 .001 32 1000 t .125 1000)

6877 /	43	=	159.93		
7040 /	45	=	156.44		
6890 /	43	=	160.23		
6837 /	37	=	184.78		
				average =	165.35

(batch-run 16 100 .001 32 1000 t .125 1000)

6081 /	22	=	276.41		
5720 /	29	=	197.24		
6148 /	31	=	198.32		
5685 /	18	=	315.83		
				average =	246.95

(batch-run 32 100 .001 32 1000 t .125 1000)

5848 /	17	=	344.00		
6037 /	18	=	335.39		
5645 /	18	=	313.61		
5610 /	11	=	510.00		
				average =	375.75

(batch-run 64 100 .001 32 1000 t .125 1000)

5669 /	13	=	436.08		
5577 /	9	=	619.67		
5683 /	11	=	516.64		
5799 /	16	=	362.44		
				average =	483.71

(batch-run 128 100 .001 32 1000 t .125 1000)

5934 /	14	=	423.86		
6104 /	9	=	678.22		
5864 /	7	=	837.71		
5767 /	11	=	524.27		
				average =	616.02

(batch-run 256 100 .001 32 1000 t .125 1000)

5774 / 11	=	524.91		
5895 / 8	=	736.88		
5994 / 13	=	461.08		
5861 / 11	=	532.82		
			average =	563.92

(batch-run 1 100 .001 35 1000 t 32 1000)

44048 / 260	=	169.42		
48468 / 280	=	173.10		
49257 / 282	=	174.67		
44790 / 253	=	177.04		
			average =	173.56

(batch-run 2 100 .001 35 1000 t 32 1000)

19043 / 161	=	118.28		
18648 / 155	=	120.31		
8818 / 63	=	139.97		
17277 / 169	=	102.23		
			average =	120.20

(batch-run 4 100 .001 35 1000 t 32 1000)

9141 / 92	=	99.36		
9892 / 104	=	95.12		
3952 / 19	=	208.00		
8103 / 91	=	89.04		
			average =	122.88

(batch-run 8 100 .001 35 1000 t 32 1000)

6472 / 59	=	109.69		
6429 / 52	=	123.63		
6093 / 46	=	132.46		
6683 / 60	=	111.38		
			average =	119.29

(batch-run 16 100 .001 35 1000 t 32 1000)

5319 / 26	=	204.58		
5069 / 23	=	220.39		
5471 / 27	=	202.63		
5893 / 25	=	235.72		
			average =	215.75

```
(batch-run 32 100 .001 32 1000 t 32 1000)
      4891 /  13  =  376.23
      4874 /  13  =  374.92
      4866 /   7  =  695.14
      4878 /  16  =  304.88
                           average =  437.79

(batch-run 64 100 .001 32 1000 t 32 1000)
      4725 /  11  =  429.55
      4787 /  10  =  478.70
      4699 /   7  =  671.29
      4830 /   9  =  536.67
                           average =  529.05

(batch-run 128 100 .001 32 1000 t 32 1000)
      4935 /  13  =  379.62
      5099 /  10  =  509.90
      4814 /   7  =  687.71
      4710 /  13  =  362.31
                           average =  484.89

(batch-run 256 100 .001 35 1000 t 32 1000)
      5394 /  13  =  414.92
      5399 /  17  =  317.59
      5043 /   8  =  630.38
      5081 /   9  =  564.56
                           average =  481.86

(batch-run 1 100 .001 158 500 t 1 5000)
    272143 /  70  =  3887.76
    278566 /  66  =  4220.70
    190987 /  47  =  4063.55
    268351 /  81  =  3312.98
                           average =  3871.25

(batch-run 2 100 .001 158 500 t 1 5000)
    116240 / 108  =  1076.30
    110374 / 140  =  788.39
     22072 /  25  =  882.88
    117633 / 101  =  1164.68
                           average =  978.06
```

(batch-run 4 100 .001 158 500 t 1 5000)

38678 / 117	=	330.58
36215 / 120	=	301.79
39379 / 111	=	54.77
44540 / 119	=	374.29
	average =	340.36

(batch-run 8 100 .001 158 500 t 1 5000)

15731 / 74	=	212.58
15011 / 85	=	176.60
14437 / 70	=	206.24
13357 / 71	=	188.13
	average =	195.89

(batch-run 16 100 .001 158 500 t 1 5000)

5447 / 53	=	102.77
6770 / 42	=	161.19
6819 / 54	=	126.28
6223 / 50	=	124.46
	average =	128.68

(batch-run 32 100 .001 158 500 t 1 5000)

2602 / 29	=	89.72
2437 / 26	=	93.73
1927 / 17	=	113.35
2947 / 34	=	86.68
	average =	95.87

(batch-run 64 100 .001 158 500 t 1 5000)

1706 / 20	=	85.30
1641 / 15	=	109.40
1618 / 18	=	89.89
1557 / 21	=	74.14
	average =	89.68

(batch-run 128 100 .001 158 500 t 1 5000)

1287 / 11	=	117.00
1060 / 9	=	117.78
1097 / 5	=	219.40
1299 / 14	=	92.79
	average =	136.74

```
(batch-run 256 100 .001 158 500 t 1 5000)
     1069 /  11  =     97.18
     1138 /   6  =    189.67
      854 /   4  =    213.50
      991 /   9  =    110.11
                           average =   152.62

(batch-run 1 100 .001 158 500 t 8 5000)
   200753 / 274  =    732.68
   203085 / 284  =    715.09
   211287 / 285  =    741.36
   204211 / 279  =    731.94
                           average =   730.27

(batch-run 2 100 .001 158 500 t 8 5000)
    77002 / 222  =    346.86
    77410 / 217  =    356.73
    78779 / 226  =    348.58
    84805 / 248  =    341.96
                           average =   348.53

(batch-run 4 100 .001 158 500 t 8 5000)
    28019 / 147  =    190.61
    25576 / 150  =    170.51
    27131 / 148  =    183.32
    28856 / 167  =    172.79
                           average =   179.31

(batch-run 8 100 .001 158 500 t 8 5000)
     3828 /  22  =    174.00
     9423 /  90  =    104.70
     9781 / 101  =     96.84
    10658 /  89  =    119.75
                           average =   123.82

(batch-run 16 100 .001 158 500 t 8 5000)
     4660 /  51  =     91.37
     4733 /  53  =     89.30
     4840 /  54  =     89.63
     4994 /  64  =     78.03
                           average =    87.08
```

```
(batch-run 32 100 .001 158 500 t 8 5000)
              3064 /  25   =    122.56
              2982 /  35   =     85.20
              3104 /  26   =    119.38
              3095 /  32   =     96.72
                                         average =    103.47

(batch-run 64 100 .001 158 500 t 8 5000)
              2726 /  21   =    129.81
              2706 /  13   =    208.15
              2794 /  18   =    155.22
              2960 /  28   =    105.71
                                         average =    149.72

(batch-run 128 100 .001 158 500 t 8 5000)
              2646 /  16   =    165.38
              2559 /  17   =    150.53
              2713 /  12   =    226.08
              2687 /  14   =    191.93
                                         average =    183.48

(batch-run 256 100 .001 158 500 t 8 5000)
              2492 /   7   =    356.00
              2689 /  11   =    244.45
              2675 /   6   =    445.83
              2517 /   8   =    314.63
                                         average =    340.23

(batch-run 1 100 .001 158 500 t .125 5000)
            211811 / 251   =    843.87
            217237 / 256   =    848.58
            195056 / 233   =    837.15
            209978 / 249   =    843.29
                                         average =    843.22

(batch-run 2 100 .001 158 500 t .125 5000)
             90538 / 184   =    492.05
             22602 /  40   =    565.05
             11504 /  19   =    605.47
             91182 / 184   =    495.55
                                         average =    539.53
```

(batch-run 4 100 .001 158 500 t .125 5000)

32614 / 125	=	260.91		
32146 / 124	=	259.24		
35781 / 131	=	273.14		
32436 / 127	=	255.40		
			average =	262.17

(batch-run 8 100 .001 158 500 t .125 5000)

13667 / 87	=	157.09		
15263 / 75	=	203.51		
15424 / 95	=	162.36		
13986 / 79	=	177.04		
			average =	175.00

(batch-run 16 100 .001 158 500 t .125 5000)

5455 / 50	=	109.10		
7265 / 52	=	139.71		
6173 / 48	=	128.60		
5989 / 48	=	124.77		
			average =	125.55

(batch-run 32 100 .001 158 500 t .125 5000)

3786 / 29	=	130.55		
4067 / 33	=	123.24		
3664 / 21	=	174.48		
3095 / 32	=	96.72		
			average =	131.25

(batch-run 64 100 .001 158 500 t .125 5000)

3404 / 18	=	189.11		
3313 / 16	=	207.06		
3019 / 14	=	215.64		
2970 / 17	=	174.71		
			average =	196.63

(batch-run 128 100 .001 158 500 t .125 5000)

2935 / 14	=	209.64		
3041 / 8	=	380.13		
3149 / 10	=	314.90		
3109 / 12	=	259.08		
			average =	290.94

```
(batch-run 256 100 .001 158 500 t .125 5000)
        3064 /   4  =   766.00
        2937 /   8  =   367.13
        2860 /   6  =   476.67
        2833 /   5  =   566.60
                        average =   544.10

(batch-run 1 100 .001 158 500 t 2 5000)
      204781 / 273  =   750.11
      208018 / 270  =   770.44
      206474 / 268  =   770.43
      203159 / 271  =   749.66
                        average =   760.16

(batch-run 2 100 .001 158 500 t 2 5000)
       80825 / 191  =   423.17
       87004 / 206  =   422.35
       87415 / 198  =   441.49
       90614 / 183  =   495.16
                        average =   445.54

(batch-run 4 100 .001 158 500 t 2 5000)
       23277 /  92  =   253.01
       36062 / 138  =   261.31
       32422 / 139  =   233.25
       33032 / 143  =   230.99
                        average =   244.64

(batch-run 8 100 .001 158 500 t 2 5000)
       13584 /  87  =   156.14
       11578 /  83  =   139.49
       14635 /  99  =   147.83
       12180 /  81  =   150.37
                        average =   148.46

(batch-run 16 100 .001 158 500 t 2 5000)
        5515 /  56  =    98.48
        5381 /  51  =   105.51
        6072 /  54  =   112.44
        5112 /  45  =   113.60
                        average =   107.51
```

(batch-run 32 100 .001 158 500 t 2 5000)

3498 / 25	=	139.92
3286 / 18	=	182.56
3632 / 29	=	125.24
3404 / 20	=	170.20
	average =	154.48

(batch-run 64 100 .001 158 500 t 2 5000)

2466 / 12	=	205.50
2151 / 17	=	126.53
2058 / 16	=	128.63
2295 / 19	=	120.79
	average =	145.36

(batch-run 128 100 .001 158 500 t 2 5000)

1431 / 10	=	143.10
1464 / 9	=	162.67
1542 / 10	=	154.20
1729 / 10	=	172.90
	average =	158.22

(batch-run 256 100 .001 158 500 t 2 5000)

1282 / 6	=	213.67
1300 / 5	=	260.00
1272 / 4	=	318.00
1218 / 3	=	406.00
	average =	299.42

(batch-run 1 100 .001 610 500 t 1 20000)

1038377 / 302	=	3438.33

(batch-run 16 100 .001 610 500 t 1 20000)

36084 / 150	=	240.56
34887 / 132	=	264.30
34661 / 147	=	235.79
21799 / 78	=	279.47
	average =	255.03

(batch-run 32 100 .001 610 500 t 1 20000)

10777 / 105	=	102.64
11188 / 97	=	115.34
9535 / 86	=	110.87
10997 / 104	=	105.74
	average =	108.65

```
(batch-run 64 100 .001 610 500 t 1 20000)
          4495 /  48  =   93.65
          4711 /  43  =  109.56
          5217 /  59  =   88.42
          4951 /  54  =   91.69
                              average =   95.83

(batch-run 256 100 .001 610 500 t 1 20000)
          2305 /  20  =  115.25
          2357 /  16  =  147.31
          2468 /  23  =  107.30
          2384 /  23  =  103.65
                              average =  118.38

(batch-run 1028 100 .001 610 500 t 1 20000)
          1863 /   8  =  232.88
          1844 /   5  =  368.80
          1826 /   6  =  304.33
          1748 /   9  =  194.22
                              average =  275.06

(batch-run 16 100 .001 610 500 t 2 20000)
         30917 / 133  =  232.46
         24650 /  38  =  648.68
         32846 / 148  =  221.93
         17795 /  76  =  234.14
                              average =  334.30

(batch-run 32 100 .001 610 500 t 2 20000)
         12175 /  98  =  124.23
         13040 /  93  =  140.22
         12808 /  84  =  152.48
         12776 /  91  =  140.40
                              average =  139.33

(batch-run 64 100 .001 630 500 t 2 20000)
          5869 /  52  =  112.87
          5799 /  48  =  120.81
          4935 /  46  =  107.28
          5978 /  54  =  110.70
                              average =  112.92
```

(batch-run 256 100 .001 630 500 t 2 20000)

3108 / 23	=	135.13	
2721 / 19	=	143.21	
2727 / 17	=	160.41	
2874 / 18	=	159.67	
		average =	149.60

(batch-run 1028 100 .001 610 500 t 2 20000)

1496 / 4	=	374.00	
1587 / 7	=	226.71	
1467 / 5	=	293.40	
1512 / 4	=	378.00	
		average =	318.03

(batch-run 16 100 .001 630 500 t 8 20000)

26518 / 156	=	169.99	
26709 / 162	=	164.87	
28783 / 184	=	156.43	
26777 / 165	=	162.28	
		average =	163.39

(batch-run 32 100 .001 630 500 t 8 20000)

11564 / 109	=	106.09	
9722 / 91	=	106.84	
9925 / 99	=	100.25	
11253 / 117	=	96.18	
		average =	102.34

(batch-run 64 100 .001 630 500 t 8 20000)

5305 / 66	=	80.38	
2855 / 23	=	124.13	
5347 / 56	=	95.48	
4984 / 52	=	95.85	
		average =	99.00

(batch-run 1028 100 .001 1525 500 t 1 50000)

3007 / 7	=	429.57

(batch-run 64 100 .001 2 8000 t 1 50)

50961 / 87	=	585.76	
51892 / 100	=	518.92	
51607 / 101	=	510.96	
51345 / 99	=	518.64	
		average =	533.57

(batch-run 256 100 .001 2 8000 t 1 50)

51071 / 104	=	491.07		
50548 / 106	=	476.87		
50776 / 109	=	465.83		
51936 / 101	=	514.22		
			average =	487.00

(batch-run 1028 100 .001 2 8000 t 1 50)

51154 / 87	=	587.98		
51021 / 91	=	560.67		
51718 / 98	=	527.73		
51277 / 114	=	449.80		
			average =	531.55

(batch-run 64 100 .001 2 8000 t 2 50)

50822 / 98	=	518.59		
51305 / 105	=	488.62		
50596 / 107	=	472.86		
51284 / 95	=	539.83		
			average =	504.98

(batch-run 256 100 .001 2 8000 t 2 50)

51090 / 91	=	561.43		
50813 / 121	=	419.94		
51212 / 118	=	434.00		
50948 / 103	=	494.64		
			average =	477.50

(batch-run 1028 100 .001 2 8000 t 2 50)

49900 / 106	=	470.75		
51023 / 115	=	443.68		
51052 / 98	=	520.94		
50295 / 98	=	513.21		
			average =	487.15

(batch-run 64 100 .001 2 4000 t .5 50)

25384 / 46	=	551.82		
25659 / 47	=	545.94		
25825 / 53	=	487.26		
25111 / 41	=	612.46		
			average =	549.37

(batch-run 256 100 .001 2 4000 t .5 50)

25181 / 44	=	572.30		
24828 / 39	=	636.62		
25571 / 52	=	491.75		
25254 / 59	=	428.03		
			average =	532.18

(batch-run 1028 100 .001 2 4000 t .5 50)

25337 / 69	=	367.20		
25190 / 49	=	514.08		
25004 / 54	=	463.04		
25812 / 45	=	573.60		
			average =	479.48

(batch-run 64 100 .001 2 4000 t 8 50) (just two)

24638 / 51	=	483.10		
24229 / 48	=	504.77		
			average =	493.94

(batch-run 1 100 .001 4 4000 t 1 100)

30830 / 207	=	148.94		
29812 / 62	=	480.84		
30309 / 184	=	164.72		
30436 / 192	=	158.52		
			average =	238.26

(batch-run 2 100 .001 4 4000 t 1 100)

27680 / 107	=	258.69		
27937 / 124	=	225.30		
28204 / 115	=	245.25		
28198 / 110	=	256.35		
			average =	246.40

(batch-run 2 100 .001 4 4000 t 4 100)

29022 / 122	=	237.89		
28618 / 106	=	269.98		
28349 / 89	=	318.53		
28782 / 111	=	259.30		
			average =	271.42

(batch-run 2 100 .001 2 4000 t 1 50)

26569 / 54	=	492.02		
25980 / 50	=	519.60		
26419 / 67	=	394.31		
25946 / 51	=	508.75		
			average =	478.67

```
(batch-run 4 100 .001 2 4000 t 1 50)
        26348 /  46   =   572.78
        25712 /  50   =   514.24
        25288 /  41   =   616.78
        25629 /  64   =   400.45
                              average =   526.06
```

Appendix C: Results of the Random Action Algorithm for Samples of Sizes 1000 and 5000

C.1. Introduction

There are two groups of data presented here. The first group comprises clusters of eight runs with sample sizes of 1000 at quantized CPU values that range from 1 to 26. The data in this group are presented chronologically, so the first clusters of four tested quantized CPU values that were powers of two (enabling me to get an overview of performance), then intermediate values, then completions of the eight run groups. The second group of data comprises four sample clusters for samples sizes of 5000. All data runs in this appendix were executed on a Sun Sparc 10 workstation (locally known as "Bayes") running Sun Solaris, with each run requiring approximately eight hours of processor time on average.

In some rare cases, runs resulted in extremely low numbers of failures for the length of run. It is not entirely unexpected that code constructed over a period of close to ten years and ported across three Lisp compilers might have occasional difficulties. In these instances I chose to fault the code, toss out the resultant data point, and redo the run. All of these data points are documented, with the redo value immediately following the item it replaces.

Understanding the results: Each cluster of data is preceded by the input batch-run pattern for that cluster. The *numeric* inputs are the only ones relevant, and from left to right they are: quantized CPU value, steps (not relevant for the Random algorithm), probability of failure (for each failure mode of a gate in the simulator), decision time,

sim-cycles (the approximate number of agent decisions made), kurtosis value, and number of samples.

The data lines are written as **cost / # failure = cost per unit failure**. Each cluster is averaged, and in the second set of 1000 sample runs the cumulative averages and standard deviations are also calculated.

C.2. Runs With 1000 Samples

(batch-run 1 100 .001 23 1000 t 1 1000)

33076 / 188 = 175.94
17359 / 78 = 222.55
20689 / 101 = 204.84
32375 / 182 = 177.88
average = 195.30

(batch-run 2 100 .001 23 1000 t 1 1000)
14009 / 114 = 122.89
13040 / 119 = 109.58
14149 / 103 = 137.37
15066 / 137 = 109.97
average = 119.95

(batch-run 4 100 .001 23 1000 t 1 1000)
8410 / 69 = 121.88
8733 / 64 = 136.45
8494 / 63 = 134.83
8139 / 53 = 153.57
average = 136.68

(batch-run 8 100 .001 23 1000 t 1 1000)
6593 / 35 = 188.37
6933 / 31 = 223.65
6691 / 25 = 267.64
6727 / 33 = 203.85
average = 220.88

(batch-run 16 100 .001 23 1000 t 1 1000)

6449 / 23 = 280.39
6502 / 25 = 260.08
6586 / 23 = 263.44
6203 / 16 = 387.81

average = 297.93

(batch-run 32 100 .001 23 1000 t 1 1000)

5914 / 9 = 657.11
5939 / 10 = 593.90
6180 / 13 = 475.38
6179 / 15 = 411.93

average = 534.58

(batch-run 64 100 .001 23 1000 t 1 1000)

5931 / 12 = 494.25
6481 / 13 = 498.54
6030 / 11 = 548.18
6090 / 10 = 609.00

average = 537.49

(batch-run 128 100 .001 23 1000 t 1 1000)

6496 / 14 = 464.00
5924 / 8 = 740.50
6136 / 11 = 557.82
6111 / 12 = 509.25

average = 567.89

(batch-run 7 100 .001 23 1000 t 1 1000)

7438 / 54 = 137.74
7144 / 38 = 188.00
7351 / 48 = 153.15
7212 / 42 = 171.71

average = 162.65

(batch-run 9 100 .001 23 1000 t 1 1000)

6350 / 22 = 288.64
6914 / 36 = 192.06
7031 / 44 = 159.80
6668 / 28 = 238.14

average = 219.66

(batch-run 10 100 .001 23 1000 t 1 1000)
6753 / 32 = 211.03
7009 / 41 = 170.95
7050 / 45 = 156.67
7155 / 42 = 170.36
average = 177.25

(batch-run 11 100 .001 23 1000 t 1 1000)
6598 / 29 = 227.52
6776 / 28 = 242.00
6580 / 24 = 274.17
6546 / 26 = 251.77
average = 248.87

(batch-run 12 100 .001 23 1000 t 1 1000)
6428 / 22 = 292.18
6464 / 20 = 323.20
6542 / 20 = 327.10
6236 / 28 = 222.71
average = 291.29

(batch-run 13 100 .001 23 1000 t 1 1000)
6332 / 26 = 243.54
6340 / 20 = 317.00
6417 / 25 = 256.68
6269 / 19 = 329.95
average = 286.79

(batch-run 14 100 .001 23 1000 t 1 1000)
6378 / 24 = 265.75
6791 / 24 = 282.96
6644 / 19 = 349.68
6206 / 32 = 193.94
average = 273.08

(batch-run 15 100 .001 23 1000 t 1 1000)
6146 / 24 = 256.08
6481 / 20 = 324.05
6320 / 26 = 243.08
6437 / 29 = 221.97
average = 261.30

(batch-run 1 100 .001 23 1000 t 1 1000)
31843 / 190 = 167.59
30881 / 171 = 180.59
32352 / 176 = 183.18
33731 / 185 = 182.33
average = 178.42

(batch-run 3 100 .001 23 1000 t 1 1000)
10487 / 85 = 123.38
11283 / 90 = 125.37
10035 / 80 = 125.44
10500 / 88 = 119.32
average = 123.38

(batch-run 5 100 .001 23 1000 t 1 1000)
8269 / 62 = 133.37
8151 / 57 = 143.00
8291 / 62 = 133.73
8272 / 57 = 145.12
average = 138.81

(batch-run 6 100 .001 23 1000 t 1 1000)
7706 / 53 = 145.40
7750 / 47 = 164.89
7313 / 44 = 166.20
7355 / 46 = 159.89
average = 159.10

(batch-run 17 100 .001 23 1000 t 1 1000)
6420 / 24 = 267.50
6537 / 20 = 326.85
6803 / 23 = 295.78
6192 / 18 = 344.00
average = 308.53

(batch-run 18 100 .001 23 1000 t 1 1000)
6399 / 25 = 255.96
6480 / 24 = 270.00
6282 / 20 = 314.10
6728 / 25 = 269.12
average = 277.30

(batch-run 19 100 .001 23 1000 t 1 1000)
6368 / 24 = 265.33
6365 / 27 = 235.74
6410 / 24 = 267.08
6438 / 19 = 338.84
average = 276.75

(batch-run 20 100 .001 23 1000 t 1 1000)
6129 / 19 = 322.58
6336 / 24 = 264.00
6900 / 28 = 246.43
6532 / 20 = 326.60
average = 289.90

(batch-run 21 100 .001 23 1000 t 1 1000)
6537 / 33 = 198.09
6721 / 34 = 197.68
6343 / 22 = 288.32
6493 / 21 = 309.19
average = 248.32

(batch-run 22 100 .001 23 1000 t 1 1000)
6492 / 15 = 432.80
6391 / 20 = 319.55
6517 / 20 = 325.85
6541 / 22 = 297.32
average = 343.88

(batch-run 23 100 .001 23 1000 t 1 1000)
5866 / 10 = 586.60
6442 / 12 = 536.83
6155 / 7 = 879.29
6232 / 16 = 389.50
average = 598.06

(batch-run 24 100 .001 23 1000 t 1 1000)
6186 / 12 = 515.50
6234 / 18 = 346.33
6547 / 15 = 436.47
6093 / 16 = 380.81
average = 419.78

(batch-run 25 100 .001 23 1000 t 1 1000)

5941 / 9 = 660.11
6223 / 10 = 622.30
6450 / 6 = 1075.00
6171 / 12 = 514.25

average = 717.92

(batch-run 26 100 .001 23 1000 t 1 1000)

6335 / 10 = 633.50
5935 / 8 = 741.88
6354 / 10 = 635.40
6285 / 14 = 448.93

average = 614.93

(batch-run 26 100 .001 23 1000 t 1 1000)

5973 / 11 = 543.00
6107 / 13 = 469.77
6373 / 10 = 637.30
6344 / 10 = 634.40

average = 571.12
CUM AVE = 593.02
SPL VAR = 98.36

(batch-run 25 100 .001 23 1000 t 1 1000)

6172 / 13 = 474.77
6280 / 9 = 697.78
6290 / 16 = 393.13
6246 / 12 = 520.50

average = 521.55
CUM AVE = 619.73
SPL VAR = 209.87

(batch-run 24 100 .001 23 1000 t 1 1000)

5955 / 15 = 397.00
6289 / 14 = 449.21
6296 / 15 = 419.73
6172 / 13 = 474.77

average = 435.18
CUM AVE = 427.48
SPL VAR = 53.84

(batch-run 23 100 .001 23 1000 t 1 1000)

6411 / 10 = 641.10
6296 / 19 = 331.37
6215 / 19 = 327.11
6009 / 11 = 546.27

average = 461.46
CUM AVE = 529.76
SPL VAR = 184.48

(batch-run 22 100 .001 23 1000 t 1 1000)

6439 / 17 = 378.76
6396 / 18 = 355.33
6188 / 19 = 325.68
6457 / 18 = 358.72

average = 354.62
CUM AVE = 349.25
SPL VAR = 42.53

(batch-run 21 100 .001 23 1000 t 1 1000)

6572 / 30 = 219.07
6499 / 20 = 324.95
6540 / 19 = 344.21
6688 / 24 = 278.67

average = 281.73
CUM AVE = 270.02
SPL VAR = 57.90

(batch-run 20 100 .001 23 1000 t 1 1000)

6379 / 24 = 265.79
6869 / 31 = 221.58
6450 / 24 = 268.75
6692 / 30 = 223.07

average = 244.80
CUM AVE = 267.35
SPL VAR = 39.77

(batch-run 19 100 .001 23 1000 t 1 1000)

6415 / 20 = 320.75
6312 / 23 = 274.43
6735 / 29 = 232.24
6600 / 27 = 244.44

average = 267.97
CUM AVE = 272.36
SPL VAR = 38.86

(batch-run 18 100 .001 23 1000 t 1 1000)
6626 / 23 = 288.09
6311 / 16 = 394.44
6498 / 26 = 249.92
6417 / 22 = 291.68

average = 306.03
CUM AVE = 291.66
SPL VAR = 46.40

(batch-run 17 100 .001 23 1000 t 1 1000)
6673 / 20 = 333.65
6426 / 20 = 321.30
6398 / 22 = 290.81
6279 / 22 = 285.41

average = 307.79
CUM AVE = 308.16
SPL VAR = 26.94

(batch-run 16 100 .001 23 1000 t 1 1000)
6474 / 26 = 249.00
6513 / 19 = 342.79
6608 / 21 = 314.67
6650 / 30 = 221.67

average = 282.03
CUM AVE = 289.98
SPL VAR = 54.78

(batch-run 15 100 .001 23 1000 t 1 1000)
6656 / 19 = 350.32
6241 / 23 = 271.34
6664 / 25 = 266.56
6629 / 23 = 288.22

average = 294.11
CUM AVE = 277.70
SPL VAR = 42.21

(batch-run 14 100 .001 23 1000 t 1 1000)
6515 / 21 = 310.24
6470 / 22 = 294.09
6463 / 21 = 307.76
6439 / 22 = 292.68

average = 301.19
CUM AVE = 287.14
SPL VAR = 44.90

(batch-run 13 100 .001 23 1000 t 1 1000)

6444 / 31 = 207.87
6348 / 19 = 334.10
6419 / 25 = 256.76
6131 / 24 = 255.46

average = 263.55
CUM AVE = 275.17
SPL VAR = 46.02

(batch-run 12 100 .001 23 1000 t 1 1000)

6768 / 22 = 307.64
6493 / 29 = 223.90
6461 / 28 = 230.75
6384 / 21 = 304.00

average = 266.57
CUM AVE = 278.93
SPL VAR = 45.39

(batch-run 11 100 .001 23 1000 t 1 1000)

6663 / 35 = 190.37
6660 / 28 = 237.86
6444 / 35 = 184.11
6574 / 24 = 273.92

average = 221.57
CUM AVE = 235.22
SPL VAR = 33.86

(batch-run 10 100 .001 23 1000 t 1 1000)

6658 / 40 = 166.45
6703 / 33 = 203.12
6578 / 31 = 212.19
6717 / 36 = 186.58

average = 192.09
CUM AVE = 184.67
SPL VAR = 21.73

(batch-run 9 100 .001 23 1000 t 1 1000)

6869 / 33 = 208.15
6661 / 32 = 208.16
6695 / 30 = 223.17
6834 / 37 = 184.70

average = 206.05
CUM AVE = 212.85
SPL VAR = 38.86

(batch-run 8 100 .001 23 1000 t 1 1000)

7059 / 40 = 176.48
6435 / 38 = 169.34
6531 / 27 = 241.89
6843 / 34 = 201.26

average = 197.24
CUM AVE = 209.06
SPL VAR = 33.55

(batch-run 7 100 .001 23 1000 t 1 1000)

7731 / 51 = 151.59
7612 / 49 = 155.35
7033 / 31 = 226.87
7065 / 44 = 160.57

average = 173.60
CUM AVE = 168.12
SPL VAR = 28.03

(batch-run 6 100 .001 23 1000 t 1 1000)

7228 / 46 = 157.13
7376 / 47 = 156.94
6772 / 27 = 250.81
7472 / 45 = 166.04

average = 182.73
CUM AVE = 170.91
SPL VAR = 33.00

(batch-run 5 100 .001 23 1000 t 1 1000)

7800 / 57 = 136.84
6614 / 4 = 1653.50 (toss out)
7643 / 50 = 152.86 (redo)
8099 / 53 = 152.81
7013 / 25 = 280.52

average = 180.76
CUM AVE = 159.78
SPL VAR = 49.39

(batch-run 4 100 .001 23 1000 t 1 1000)

8799 / 68 = 129.40
9018 / 74 = 121.86
8696 / 59 = 147.39
9199 / 62 = 148.37

average = 136.76
CUM AVE = 136.72
SPL VAR = 12.14

(batch-run 3 100 .001 23 1000 t 1 1000)
9605 / 86 = 111.69
10301 / 80 = 128.76
10218 / 83 = 123.11
10548 / 85 = 124.09

average = 121.91
CUM AVE = 122.65
SPL VAR = 5.16

(batch-run 2 100 .001 23 1000 t 1 1000)
14556 / 116 = 125.48
12576 / 96 = 131.00
13289 / 102 = 130.28
14437 / 123 = 117.37

average = 126.03
CUM AVE = 122.99
SPL VAR = 10.08

(batch-run 1 100 .001 23 1000 t 1 1000)
30252 / 176 = 171.89
31338 / 180 = 174.10
24583 / 105 = 234.12
34152 / 187 = 182.63

average = 190.69
CUM AVE = 192.99
SPL VAR = 24.30

C.3. Runs With 5000 Samples

(batch-run 1 100 .001 158 500 t 1 5000)
195911 / 301 = 650.87
209680 / 277 = 756.97
13425 / 14 = 958.93
206001 / 275 = 749.09

average = 778.96

(batch-run 2 100 .001 158 500 t 1 5000)
83471 / 210 = 397.48
81210 / 238 = 341.22
80523 / 229 = 351.63
39177 / 108 = 362.75

average = 363.27

(batch-run 4 100 .001 158 500 t 1 5000)
30432 / 137 = 222.13
25503 / 120 = 212.53
28470 / 153 = 186.08
31176 / 155 = 201.14
average = 205.47

(batch-run 8 100 .001 158 500 t 1 5000)
12187 / 94 = 129.65
12510 / 104 = 120.29
12845 / 95 = 135.21
14465 / 100 = 144.65
average = 132.45

(batch-run 16 100 .001 158 500 t 1 5000)
6920 / 52 = 133.08
6435 / 47 = 136.91
7625 / 57 = 133.77
7266 / 64 = 113.53
average = 129.32

(batch-run 32 100 .001 158 500 t 1 5000)
4611 / 2 = 2305.50 (toss out)
4879 / 30 = 145.97 (redo)
5073 / 31 = 163.65
5007 / 32 = 156.47
5386 / 35 = 153.89
average = 154.99

(batch-run 64 100 .001 158 500 t 1 5000)
4561 / 18 = 253.39
4732 / 13 = 364.00
4804 / 12 = 400.33
4836 / 24 = 201.50
average = 304.81

(batch-run 128 100 .001 158 500 t 1 5000)
4395 / 10 = 439.50
4448 / 6 = 741.33
4195 / 7 = 599.29
4338 / 11 = 394.36
average = 543.62

(batch-run 64 100 .001 105 500 t 1 5000)
2925 / 8 = 365.62

C.4. Calculation of Means and Confidence Intervals for Samples of Size 1000

Quantized CPU Value	Cost Per Failure	Standard Deviation	Lower Bd. Confidence	Upper Bd. Confidence
26.0000	593.0200	98.0600	543.9627	642.0773
25.0000	619.7300	209.8700	514.7366	724.7234
24.0000	427.4800	53.8400	400.5450	454.4150
23.0000	529.7600	184.4800	437.4687	622.0513
22.0000	349.2500	42.5300	327.9732	370.5268
21.0000	270.0200	57.9000	241.0539	298.9861
20.0000	267.3500	39.7700	247.4539	287.2461
19.0000	272.3600	38.8600	252.9192	291.8008
18.0000	291.6600	46.4000	268.4471	314.8729
17.0000	308.1600	26.9400	294.6825	321.6375
16.0000	289.9800	54.7800	262.5748	317.3852
15.0000	277.7000	42.2100	256.5833	298.8167
14.0000	287.1400	44.9000	264.6775	309.6025
13.0000	275.1700	46.0200	252.1472	298.1928
12.0000	278.9300	45.3900	256.2224	301.6376
11.0000	235.2200	33.8600	218.2806	252.1594
10.0000	184.6700	21.7300	173.7990	195.5410
9.0000	212.8500	38.8600	193.4092	232.2908
8.0000	209.0600	33.5500	192.2757	225.8443
7.0000	168.1200	28.0300	154.0972	182.1428
6.0000	170.9100	33.0000	154.4008	187.4192
5.0000	159.7800	49.3900	135.0713	184.4887
4.0000	136.7200	12.1400	130.6466	142.7934
3.0000	122.6500	5.1600	120.0686	125.2314
2.0000	122.9900	10.0800	117.9472	128.0328
1.0000	192.9900	24.3000	180.8332	205.1468

Table C.1: Calculation of Means and Confidence Intervals for Samples of Size 1000

Appendix D: Code for Calculations Involving One Submodel of the Complete Stochastic Automata Model for the OLMA Running the Random Algorithm

D.1. Matlab Code for Calculating the Stochastic Automata Network

The code presented in Figure D.1 is strictly an example of the code needed to calculate the tensor products and some critical values when the agent and simulator run at the same speed. Each agent matrix may require several such calculations, depending on the desired results. The loops implement the probability transference representing the interaction between the agent and the gate components of the simulator. This procedure is significantly simpler than the standard calculations of the discrete tensor product.

```
format long;

GATE1 = [ ...
       0.997  0.003
       0.003  0.997 ];

GATE2 = GATE1;

GATE3 = GATE2;

GATE4 = GATE3;

AGENT1 = [ ...
   0.142857 0.142857 0.142857 0.142857 0.142857 0.285714
   0.142857 0.142857 0.142857 0.142857 0.142857 0.285714
   0.142857 0.142857 0.142857 0.142857 0.142857 0.285714
   0.142857 0.142857 0.142857 0.142857 0.142857 0.285714
   0.142857 0.142857 0.142857 0.142857 0.142857 0.285714
   0.142857 0.142857 0.142857 0.142857 0.142857 0.285714 ];

FOURGATES = kron(GATE1,kron(GATE2,kron(GATE3,GATE4)));

TENS = kron(AGENT1,FOURGATES);

TENS1 = TENS;
```

```
 for Ag = 0:3
  for Gone = 0:1
   for Gtwo = 0:1
    for Gtre = 0:1
     for Gfor = 0:1
      if (Ag == 0) & (Gone == 1)
       for IAg = 0:5
           TEMP = TENS1(indexer(IAg,Gone,Gtwo,Gtre,Gfor),indexer(Ag,Gone,Gtwo,Gtre,Gfor));
           TENS1(indexer(IAg,Gone,Gtwo,Gtre,Gfor),indexer(Ag,0,Gtwo,Gtre,Gfor)) =
TENS1(indexer(IAg,Gone,Gtwo,Gtre,Gfor),indexer(Ag,0,Gtwo,Gtre,Gfor)) + TEMP;
           TENS1(indexer(IAg,Gone,Gtwo,Gtre,Gfor),indexer(Ag,Gone,Gtwo,Gtre,Gfor)) = 0;
       end
      elseif (Ag == 1) & (Gtwo == 1)
       for IAg = 0:5
           TEMP = TENS1(indexer(IAg,Gone,Gtwo,Gtre,Gfor),indexer(Ag,Gone,Gtwo,Gtre,Gfor));
           TENS1(indexer(IAg,Gone,Gtwo,Gtre,Gfor),indexer(Ag,Gone,0,Gtre,Gfor)) =
TENS1(indexer(IAg,Gone,Gtwo,Gtre,Gfor),indexer(Ag,Gone,0,Gtre,Gfor)) + TEMP;
           TENS1(indexer(IAg,Gone,Gtwo,Gtre,Gfor),indexer(Ag,Gone,Gtwo,Gtre,Gfor)) = 0;
       end
      elseif (Ag == 2) & (Gtre == 1)
       for IAg = 0:5
           TEMP = TENS1(indexer(IAg,Gone,Gtwo,Gtre,Gfor),indexer(Ag,Gone,Gtwo,Gtre,Gfor));
           TENS1(indexer(IAg,Gone,Gtwo,Gtre,Gfor),indexer(Ag,Gone,Gtwo,0,Gfor)) =
TENS1(indexer(IAg,Gone,Gtwo,Gtre,Gfor),indexer(Ag,Gone,Gtwo,0,Gfor)) + TEMP;
           TENS1(indexer(IAg,Gone,Gtwo,Gtre,Gfor),indexer(Ag,Gone,Gtwo,Gtre,Gfor)) = 0;
       end
      elseif (Ag == 3) & (Gfor == 1)
       for IAg = 0:5
           TEMP = TENS1(indexer(IAg,Gone,Gtwo,Gtre,Gfor),indexer(Ag,Gone,Gtwo,Gtre,Gfor));
           TENS1(indexer(IAg,Gone,Gtwo,Gtre,Gfor),indexer(Ag,Gone,Gtwo,Gtre,0)) =
TENS1(indexer(IAg,Gone,Gtwo,Gtre,Gfor),indexer(Ag,Gone,Gtwo,Gtre,0)) + TEMP;
           TENS1(indexer(IAg,Gone,Gtwo,Gtre,Gfor),indexer(Ag,Gone,Gtwo,Gtre,Gfor)) = 0;
       end
      end
     end
    end
   end
  end
 end
end


[p,d] = eig(TENS1');
diag(d)

p(:,1)/sum(p(:,1))


probgood = 0;
for Gate1Finder = 0:5
 NewSum(Gate1Finder + 1) = 0;
 probgood = probgood + result((Gate1Finder*16) + 1);
 for ResI = 9:16
    NewSum(Gate1Finder + 1) = NewSum(Gate1Finder + 1) + result(ResI + (16 * Gate1Finder));
 end
 NewSum(Gate1Finder + 1)
end
```

```
err = sum(NewSum)

fourerr = 4 * sum(NewSum)

probgood
```