MEASURE-ADAPTIVE STATE-SPACE CONSTRUCTION METHODS

by

Walter Douglas Obal II

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 9 8

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Final Examination Committee, we certify that we have read the dissertation prepared by **Walter Douglas Obal II**
entitled

Measure-Adaptive State-Space Construction Methods

and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of **Doctor of Philosophy**

Prof. Michael L. Marcellin	Date
Prof. William H. Sanders	Date
Prof. Steven L. Dvorak	Date
Prof. Emmanuel Fernandez	Date
Prof. Jeffrey B. Goldberg	Date

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copy of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Thesis Director Prof. Michael L. Marcellin	Date

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED:_____

ACKNOWLEDGMENTS

First and foremost, I want to acknowledge my wife, Leticia, whom I dearly love. I could never have done this without your love, help, understanding, and constant encouragement. Together we have been through much, but now that this chapter is coming to an end, we have some wonderful things to look forward to. My family, and in particular my parents, were another great source of encouragement and did much to help me through the down times.

I also want to acknowledge Bill Sanders for his patience and his willingness to stick with me through several cycles of ups and downs. His constant optimism and encouragement helped me get through the down cycles.

Latha Kant, Anand Kuratti, Luai Malhis, Aad van Moorsel, and Akber Qureshi, — thanks for your support and the encouragement you gave me. As the last graduating member of the old Performability Modeling Research Lab, I take great pleasure in joining the PMRL alumni already in commercial research and development positions. I don't think I will ever again encounter such an intelligent, diverse and interesting group of people with the level of comraderie we had in the PMRL.

I also want to acknowledge the group at the University of Illinois, especially Jay Doyle, G. P. Kavanaugh, John Sowder and Alex Williamson, for their friendly conversation and comic relief. Whenever I needed a break, I could always find entertainment in their room, where software development was always mixed with day-trading and the latest in gaming technology. For most of my visit to Illinois, I did not have a car, so I am grateful to Alex for the many rides to Meijer and the laundromat. The other members of the group, Michel Cukier, David Daly, Dan Deavours, Harpreet Duggal, David Henke, Jessica Pistole, Jennifer Ren, Paul Rubel, Chetan Sabnis, Aaron Stillman, and Patrick Webster, were also good to me, and I enjoyed the friendly rivalry between the AQuA and Möbius projects.

On the technical side, I acknowledge the following contributions. I owe the designers and implementors of the GAP software a debt of gratitude, since GAP greatly facilitated my experimentation with group theory. Dan Deavours helped by giving me his SPN state space construction code, which served as a convenient starting point for my prototype implementations of my ideas, and I had several helpful discussions with G. P. Kavanaugh on the nature of path-based performance measures.

Jenny Applequist deserves my thanks for everything she did to help me during my extended visit at the University of Illinois, but particularly for proof-reading my

dissertation. Her comments and suggestions did much to improve the readability of this dissertation.

Special thanks go to Prof. Michael Marcellin, who took over as dissertation director when Bill moved to Illinois. In addition, I thank the other members of my committee, Prof. Steven Dvorak, Prof. Emmanuel Fernandez and Prof. Jeffrey Goldberg, for their flexibility during the process of scheduling my defense. Finally, Tami Whelan, the Arizona ECE Department Graduate Student Advisor, deserves to be recognized for all she did to help make sure I actually graduated.

Former Arizona ECE Department Head Prof. Ken Galloway, current Department Head Prof. John Reagan, Associate Department Head Prof. Glen Gerhard, Prof. Michael Marefat and Nick Nelson all have my gratitude for their support during my shuttling back and forth between Arizona and Illinois. They allowed me to retain several workstations and a workspace for my research at Arizona following Bill's departure for Illinois.

Of course, I am also very grateful for the generous support of the various organizations that sponsored my research. The Defense Advanced Research Projects Agency, Information Technology Office, funded me through contract DABT63-C-0069. Renee Langefels, Peter Alejandro and the rest of the Motorola Space Systems Technology Group, deserve thanks for their long-term funding of the *UltraSAN* project. Their large-scale simulation models did much to drive improvements in the *UltraSAN* simulators. Finally, Steve West and the IBM Storage Systems Division also funded my work on *UltraSAN*. IBM and Steve West in particular made a special effort to fully understand and use the software. My collaboration with Steve led to many improvements in the software and I consider myself very fortunate to have had that experience.

To Leticia

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Much work has been done on the problem of stochastic modeling for the evaluation of performance, dependability and performability properties of systems, but little attention has been given to the interplay between the model and the performance measure of interest. Our work addresses the problem of automatically constructing Markov processes tailored to the structure of the system and the nature of the performance measures of interest. To solve this problem, we have developed new techniques for detecting and exploiting symmetry in the model structure, new reward variable specification techniques, and new state-space construction procedures. We propose a new method for detecting and exploiting model symmetry in which 1) models retain the structure of the system, and 2) all symmetry inherent in the structure of the model can be detected and exploited for the purposes of state-space reduction. Then, we extend the array of performance measures that may be derived from a given system model by introducing a class of path-based reward variables, which allow rewards to be accumulated based on sequences of states and transitions. Finally, we describe a new reward variable specification formalism and state-space construction procedure for automatically computing the appropriate level of state-space reduction based on the nature of the reward variables and the structural symmetry in the system model.

CHAPTER 1
Introduction

1.1 Background

Model-based evaluation is a useful tool that can be applied to a broad array of problems that arise in the design and operation of computer systems and networks. In the initial stage of system specification, modeling studies can aid in the development of the architecture by identifying the weak links or bottlenecks in the proposed architectures. Such knowledge is important since it allows a design team to focus its resources on the parts of the system that are crucial to its success. In the design phase, models can be used to assess and rank design choices according to one or more metrics. For existing systems, models are created to understand behavior, and to predict the impact of changes in the system design, implementation or operating policy. In this case, modeling is often combined with measurement, since data from the system can and should be used to calibrate the model.

Much work has been done on the problem of stochastic modeling for the evaluation of performance, dependability and performability properties of systems. (See [4, 43, 56, 57] for definitions and early work related to dependability and performability.) Initially, such models were relatively small and admitted closed-form solutions. However, it was soon realized that these techniques can and should be applied to more complicated systems, and the need for high-level modeling formalisms and software tools became clear.

Among the various modeling formalisms developed, stochastic extensions to Petri nets are among the more popular. Petri nets [72] were originally developed as an untimed, nondeterministic, nonstochastic modeling formalism for the purposes of proving properties of distributed systems. Petri nets were popular because of the relative ease with which synchronization and competition for resources could be represented. They were particularly useful for modeling distributed system protocols and deciding (in the formal sense) whether the protocol could deadlock.

The seminal work on stochastic extensions to Petri nets was carried out by Natkin [63] in the context of dependability evaluation, and Molloy [60] for the purposes of performance evaluation. These authors created "stochastic Petri nets" by associating probability distributions with the time between the enabling and firing times of each transition. It was then shown that if those times were exponentially distributed, the execution of the net could be represented by a Markov process. By evaluating the steady-state or transient state occupancy probabilities of the Markov process, one could derive various performance or dependability measures. At this point, the Petri net was used simply as a convenient representation of the Markov process.

Stochastic Petri nets (SPN) became popular because they provided a relatively compact representation of large Markov processes. A myriad of extensions to the basic SPN have been proposed, and tools (including our own — see Appendix B) have been developed to support modeling using the different formalisms. The interested reader is directed to the good surveys of modeling tools, such as [34], [36], or [94]. Most of these modeling tools automatically generate state-level representations of Markov processes from a high-level modeling formalism, and then solve for the state occupancy probabilities of the Markov process. Some tools also support simulation, in case models are too large or are not amenable to representation as a Markov process.

By design, high-level modeling formalisms facilitate the expression of complicated states and large sets of states. The result is that Markov processes constructed from these model specifications can be very large, and it is easy to overwhelm the resources of a modern workstation. This "state-space explosion" problem has been discussed, and solutions have been proposed, by many authors (see the introduction in Chapter 2 for a survey). Almost all authors have focused on developing techniques that reduce the number of states required to model the system structure and related behavior.

The fact is that in many models, most of the complexity is due to the nature of the desired measure, rather than to the actual system structure. With a few exceptions [57, 38], researchers have paid little attention to the interplay between the model and the measure of interest. In [57], Meyer introduces the "capability function" to relate low-level system behavior to user-oriented performance levels. In [38], Haverkort discusses the importance of specifying performance, dependability and performability measures at the appropriate level within modeling tools. Zeigler's notion of the "experimental frame," which he developed as part of his theory of simulation [96, 97, 98], is also germane, although his application of this concept to the modular specification and structural simplication of simulation models is very different from our state-space reduction goals.

The relationship between the measure of interest and the required state-space size is very important, since it is the measure of interest that usually determines the level of detail required in a model, which ultimately determines the number of states in a Markov process representation of the model.

1.2 Research Objectives

The purpose of this research is to develop methods for detecting and exploiting properties of models and measures to automatically construct Markov processes

that are tailored to the measure of interest as well as the structure of the system. We call this approach "measure-adaptive state-space construction." To achieve this goal, we develop new techniques for detecting and exploiting symmetry, new reward variable specification techniques, and new state-space construction procedures.

In Chapter 2, we propose a new method for detecting and exploiting model symmetry in which 1) models retain the structure of the system, and 2) all symmetry inherent in the structure of the model can be detected and exploited for the purposes of state-space reduction. Then, in Chapter 3, we extend the array of performance measures that may be derived from a given system model by introducing a class of path-based reward variables, which allow rewards to be accumulated based on sequences of states and transitions. Finally, in Chapter 4, we describe a new reward variable specification formalism and state-space construction procedure for automatically computing the appropriate level of state-space reduction based on the nature of the reward variables and the structural symmetry in the system model.

CHAPTER 2
Detecting and Exploiting Model Symmetry

2.1 Introduction

In this chapter we develop a new technique for detecting and exploiting model symmetry. Using a simple abstract modeling formalism, we define models and composed models, and show how to use the information embedded in the composed model to detect and exploit symmetries for the purpose of state-space reduction. The theory developed in this chapter is an important component of measure-adaptive state-space construction, since it yields the structural restrictions on state-space reduction.

2.2 Background

The question of how to cope with large state spaces has received much attention from the modeling community over the last two decades. The various solutions fall into two categories. Each approach either seeks to tolerate large state spaces, or seeks to reduce large state spaces to smaller ones. Research on the problem of tolerating very large state spaces has focused on efficient algorithms and data structures in an effort to maximize the number of states that can be represented and the speed with which they may be manipulated within the memory hierarchy of a workstation. Examples of this approach include the Kronecker product algorithms of Plateau [73, 74], Buchholz [10], Ciardo et al. [19, 20], Donatelli [28],

and Kemper [41], their recent collaboration [9], and the methods of Deavours and Sanders [27, 26]. Methods for partial exploration of the state-space with error bounds for some measures are discussed in [25, 37, 13]. The new approach we describe in this chapter falls into the second category.

Research on state-space reduction has focused on methods for exploiting the structure of the model. Examples of this approach are the papers of Aupperle and Meyer [1, 2], the hierarchical modeling techniques of Buchholz [8], Carrasco's work with stochastic high-level Petri nets [12], stochastic well-formed nets [16], performance evaluation process algebra [39], reduced base model construction [85], and the symmetry exploitation algorithm of Somani [92]. These approaches all use symmetry to reduce the state space by lumping together states that correspond to symmetric configurations. Alternatives to these exact methods are the decomposition method of Ciardo and Trivedi [21], which treats nearly independent submodels as independent and uses fixed-point iteration to solve the model, and the bounds on "quasi-lumpable" models described by Franceschinis and Muntz [32, 33].

Each symmetry exploitation technique has its advantages and disadvantages. Some techniques are easy to use but limited in the types of symmetry that can be detected. Other techniques do well at detecting symmetry, but the specification language leads to models that are difficult to read. Ideally, we would like to have a modeling technique that produces a model that reflects the structure of the system we are modeling, but is still able to detect and exploit all symmetry in the model. In order to reflect the structure of the system, the method needs to be compositional, with explicit relationships between submodels. To exploit symmetry, the model specification must either directly indicate the symmetry that is present or be amenable to an analysis that detects the symmetry.

In this chapter we present a new technique for detecting and exploiting symmetry in discrete-state Markov models. The technique relies on a composition graph that

specifies interaction between submodels, and automatically detects all symmetry present in the graph structure. We present the technique in the general context of discrete-state Markov models, since our work is not specific to any existing modeling formalism. The theory underlying our approach uses results from group and graph theory, which provide a rigorous foundation for the work. With our technique, models retain the structure of the system, and all symmetry in the model is detected and exploited to reduce the state space. Using a prototype implementation of the procedures presented in this chapter, we reduced the size of the state space by several orders of magnitude for several examples.

2.3 Model Description

The model specification formalism is meant to simplify the exposition by providing the minimum notation needed to discuss the composition of models and the construction of the underlying stochastic process. Many different formalisms for describing discrete event systems can be mapped onto this basic notation, but the ideas presented here are useful regardless of the details of the specification.

Definition 1 *A* model *is a five-tuple* $(S, E, \varepsilon, \lambda, \tau)$ *where*

- S *is a set of state variables* $\{s_1, s_2, \ldots, s_n\}$ *that take values in* $\mathbb{N}$*, the set of nonnegative integers. The state of the model is defined as a mapping* $\mu : S \to \mathbb{N}$*, where for all* $s \in S$*,* $\mu(s)$ *is the value of state variable* s*. Let* $M = \{\mu \mid \mu : S \to \mathbb{N}\}$ *be the set of all such mappings.*

- E *is the set of events that may occur.*

- $\varepsilon : E \times M \to \{0, 1\}$ *is the event enabling function. For each* $e \in E$ *and* $\mu \in M$*,* $\varepsilon(e, \mu) = 1$ *if event* e *may occur when the current state of the model is* μ*, and zero otherwise.*

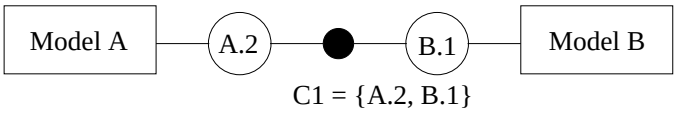


Figure 2.1: Models are connected through shared state variables.

- $\lambda : E \times M \rightarrow (0, \infty)$ *is the transition rate function. For each event* e *and state* μ *such that* $\varepsilon(e, \mu) = 1$*, event* e *occurs with rate* $\lambda(e, \mu)$ *while in state* μ*.*

- $\tau : E \times M \rightarrow M$ *is the state transition function. For each* $e \in E$ *and* $\mu \in M$*,* $\tau(e, \mu) = \mu'$*, the new state of the model that is reached when* e *occurs in* μ*.*

The *behavior of a model* is a characterization of possible sequences of events and states. Event occurrence rates are determined by λ. In Definition 1, once an event is chosen, the next state is determined by τ. Given that the current state of the model is μ, the probability of transition to a particular next state, μ', is the probability that the next event to occur is such that $\tau(e, \mu) = \mu'$. This is calculated as

$$\Pr\{\mu \rightarrow \mu'\} = \frac{\sum_{\{e \in E | \tau(e,\mu) = \mu'\}} \lambda(e, \mu)}{\sum_{\{e \in E | \varepsilon(e,\mu) = 1\}} \lambda(e, \mu)}.$$

Models are connected together through shared state variables to form "composed models." Figure 2.1 shows an example where two models are composed by specifying the superposition of two state variables. Models A and B each have state variable sets containing two state variables $\{A.1, A.2\}$ and $\{B.1, B.2\}$. In this case, the second state variable for instance A is joined to the first state variable for instance B, forming a single composed model state variable named $C1$. The resulting composed model state variable set $S = \{A.1, C1, B.2\}$. As shown in Figure 2.1, $C1$ is the connection representing the superposition of $A.2$ and $B.1$.

Systems composed of multiple identical subsystems exhibit symmetry. For example, consider Figure 2.2, which shows a ring of dual-processor nodes. Each of the boxes labeled "R" represents a network node, and each box labeled "P" represents a processor. Figure 2.2 serves well to demonstrate symmetry ideas that will

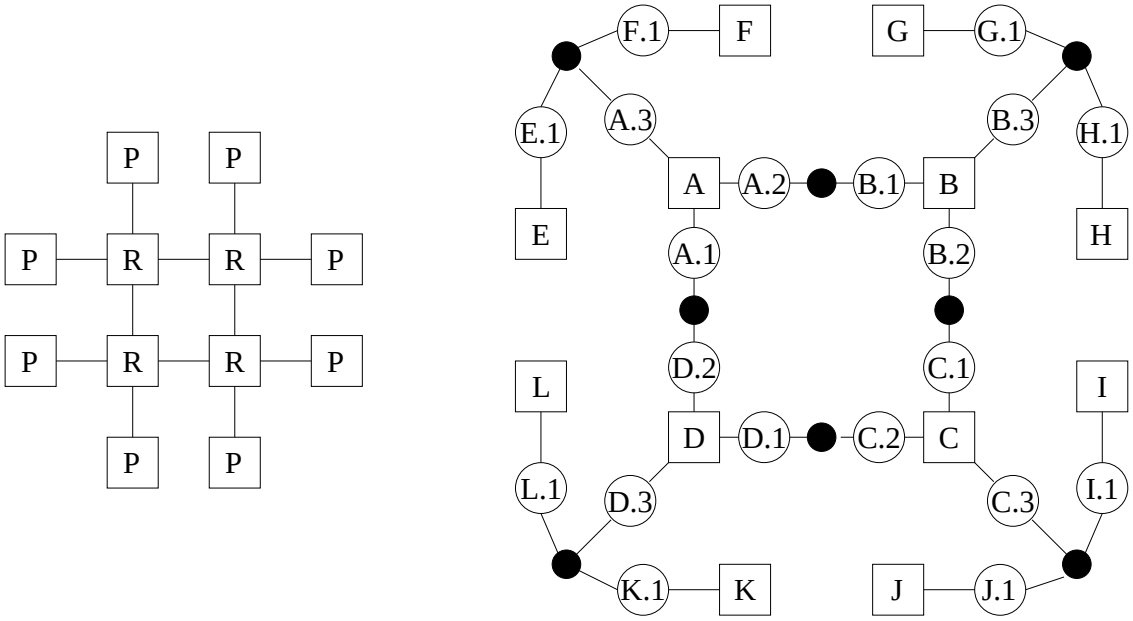


Figure 2.2: (a) Ring of dual processors system and (b) model composition graph

be more completely exploited in Section 2.8. For this system, we make two models, one for the network node and one for the processor, and then build the composed model from "instances" of these models. Before showing how this is done, we give the formal definition of a composed model.

Definition 2 *A* composed model *is a four-tuple* (Σ, I, κ, C) *where*

- Σ *is a set of models.*
- I *is a set of instances of models in* Σ*. Each instance is a complete copy of a model in* Σ*, and is independent of all other instances, except as explicitly defined through the connection set.*
- $\kappa : I \rightarrow \Sigma$ *is the instance type function.*

- *C is a set of connections. Each connection $c \in C$ represents a state variable shared among two or more instances. In this way, c represents the superposition of one state variable from each connected instance. The element c specifies the set of instance state variables it represents.*

A composed model thus partitions the state variable set of each instance into two subsets—those variables that are shared with other instances, and those that are not. Subsets of a state variable set will be called *state variable fragments*. The subset of an instance state variable set that is not shared will be called the *private state variable fragment* of that instance. Each state variable that is not in the private state variable fragment is shared, and appears as an element in exactly one connection set.

The tuple notation in Definition 2 is useful for formal definition, but the graphical representation illustrated in Figure 2.1 is better suited for visualization of the structure of a composed model. The following conventions for drawing composed models will be adopted. The private state variable fragment for an instance is depicted by a box with the instance identifier, while shared state variable fragments are represented by circles labeled with a fragment identifier comprising the instance name and state variable identifier. Connection nodes are represented by small solid circles.

We call the graphical representation a "model composition graph." A *model composition graph* is an undirected graph, $G = (V, W)$. Elements of the vertex set, V, are private state variable fragments, shared variables or connection nodes. Every instance has exactly one private state variable fragment, but this fragment may be empty. It is possible that all state variables in an instance state variable set are shared. In this case, the empty private state fragment serves as an anchor for the shared variables, as will be understood from the requirements on the edge set. There are two rules that must be satisfied by the edge set, W, of the graph.

First, every shared state variable for an instance must be adjacent to the private fragment for that instance. Second, each shared variable is adjacent to exactly one connection node.

To elucidate this approach to modeling, we use the example of Figure 2.2. The first step in using our approach is to model the two components used in this system. We give detailed examples of component models in Section 2.8; our main focus here is the composition of models. Given models of a processor and a network node, the question is how to compose them to form the system model. Figure 2.2-b shows a model composition graph for the system. Instances A, B, C and D are instances of the network node model, while instances E–L are instances of the processor model. In drawing this model composition graph, we have assumed that the ring has direction and the processors share an interface. Thus, network node instance A has three shared state fragments. $A.1$ is A's incoming link, $A.2$ is its outgoing link, and $A.3$ is its processor interface. To form the ring, A connects its outgoing link to $B.1$, the incoming link of instance B. Meanwhile, processor instance E has a network interface, $E.1$, which is connected to $A.3$, as is $F.1$, the network interface of processor instance F. This connection indicates symmetric access to the network node. From Figure 2.2, one can see the symmetry that can be exploited. There is a rotational symmetry around the center of the ring, and the processors at each node are symmetric about the interface.

We can now describe the composed model in terms of the models it comprises. The composed model state variable set may be derived from the vertex set of the model composition graph by deleting vertices corresponding to shared state variables. The resulting composed model state variable set contains the state variables in the private state variable fragment of each instance, and a state variable for each vertex corresponding to a connection. As it was with models, the *composed model*

state is defined as a mapping $\mu : S \rightarrow \mathbb{N}$ from the composed model state variable set (S) to the nonnegative integers. M again represents the set of all possible states.

The composed model event set is simply the union of the event set for each instance. The subset of the composed model event set that originates in an instance A is denoted E_A. The event enabling function for the composed model is also a simple union of the functions for each instance. That is, given a composed model state, μ, and some event, e, the composed model event enabling function is $\varepsilon(e, \mu) = \varepsilon_A(e, \mu_A)$, when $e \in E_A$. Likewise, the composed model event rate function is $\lambda(e, \mu) = \lambda_A(e, \mu_A)$ when $e \in E_A$.

The interaction between instances in the composed model is captured in the "composed model transition function," whose definition utilizes the notion of the "local state" of an instance. The *local state of an instance* is the projection of the composed model state onto the state variables of the instance. Note that the private state variable fragment of an instance is represented explicitly in the composed model state. The shared state variables of an instance are assigned the values held by the associated connections in the composed model state. Given a composed model state μ, the local state of instance A will be denoted μ_A.

Definition 3 *The* composed model state transition function *is defined as* $\tau : E \times M \rightarrow M$*, where* E *and* M *are the set of events and the set of all possible states for the composed model. Let* τ_A *denote the state transition function for the instance* A*. Then for all* $e \in E_A$ *and composed model states* μ*,*

$$\tau(e, \mu) = (\mu - \mu_A) \cup \tau_A(e, \mu_A).$$

With the composed model functions defined, writing a procedure that will generate the state space for a composed model is straightforward, as shown in Figure 2.3. However, the detailed state space will be very large for most composed models. Fortunately, in many cases the detailed state space contains much more information

U : Unexplored states, S : Discovered states
E : Event set, E_i : Event set of instance $i \in I$

Initial state μ_0
$U = \{\mu_0\}$
$S = \{\mu_0\}$
while $U \neq \emptyset$
    choose a $\mu \in U$
    $U = U - \{\mu\}$
    $E(\mu) = \{e \in \bigcup_{i \in I} E_i \mid \varepsilon(e, \mu) = 1\}$
    for each $e \in E(\mu)$
        $\mu' = \tau(e, \mu)$
        if $\mu' \notin S$
            $S = S \cup \mu'$
            $U = U \cup \mu'$
        add arc from μ to μ' with rate $\lambda(e, \mu)$
    end for
end while

Figure 2.3: Procedure for detailed state generation from a composed model

than is needed to evaluate the dependability measure of interest. In such cases, the specific identity of a model is not required. Sometimes all that is needed is the quantity of each type of model in each state possible for that type of model. In other cases, it is not enough to know the numbers—one also needs to know something about the configuration of the various model states. An example of such a system is the BIBD network proposed by Aupperle and Meyer [1]. In either case, and in other situations where the precise identity of each component in a redundant set is not required, there are symmetries that may be exploited. Detecting such symmetries in the model is the topic of the following section.

2.4 Detecting Symmetry

In the composed model formalism of Section 2.3, the structure of the model is exposed in the model composition graph. We now describe a new method for detecting symmetry using the model composition graph. In this new approach, we use the automorphism group of the model composition graph to detect structural symmetry. The main result of this section is a proof that we can construct a Markov process with states that are the partition of the state space induced by the automorphism group of the model composition graph.

A structural symmetry is present whenever there are multiple instances of the same model present in the composed model, and the names of these instances can be permuted in some way so that the model is structurally indistinguishable from the original. A behavioral symmetry is present if the permutation of instance names can be done without changing the behavior of the model. The main tool for detecting structural symmetry in the model composition graph is the automorphism group of the graph. An automorphism of a graph permutes the names of the vertices in such a way as to maintain the structure of the graph, thus exposing a structural symmetry. We now turn to the technical details. Then we will show that the

structural symmetry induces a behavioral symmetry, which can be exploited to obtain a smaller Markov process representation.

The assumption of a homogeneous vertex set is implicit in the standard definition of graph automorphism, so that any two vertices with the same degree may be mapped onto one another. However, the model composition graph may include vertices representing instances of different models, in which case it will not have a homogeneous vertex set. In this case, automorphisms must be restricted to permutations that map vertices representing state variables of each model type only among themselves.

Let $\Xi = \{\xi_1, \xi_2, \ldots, \xi_n\}$, be a partition of V that satisfies the following requirements:

1. Two vertices are in the same partition element if and only if the vertices correspond to the same state variable fragment of instances of the same model.

2. All vertices corresponding to connection nodes are in the same partition element.

Let Γ be the automorphism group of the graph with respect to Ξ. By this it is meant that Γ is a permutation group on the vertex set of the composition graph, such that for all $\gamma \in \Gamma$, $v \in \xi_i$ if and only if $\gamma(v) \in \xi_i$.

Permutations in Γ map V onto itself. For convenience, the permutation notation $\gamma(\cdot)$ will be overloaded so it can be used with an argument that is either a state variable fragment or a state variable. This overloading is justified by the fact that elements of Γ are restricted by definition to map vertices within their own partition elements, which means that for all $\gamma \in \Gamma$, $\gamma(v)$ is a vertex with the same structure as v. Therefore, $\gamma(s)$ will be used to denote the state variable in the fragment $\gamma(v)$ that is the image of s under γ. An example should help clarify this notion. Suppose $v_1 = \{A.1, A.2\}$ is the private state variable fragment of instance A, and

$v_2 = \{B.1, B.2\}$ is the private state variable fragment of instance B. If $\gamma(v_2) = v_1$, then by definition $\gamma(B.1) = A.1$ and $\gamma(B.2) = A.2$. The next step is to demonstrate that such structural symmetries induce behavioral symmetry, and to characterize the nature of the behavioral symmetry.

To demonstrate the behavioral symmetry among symmetric structural configurations of a composed model, we must investigate the effect of an automorphism on the composed model state. An automorphism is a renaming of instances, and a composed model state is a mapping of instance state variables to numbers. One way to visualize the effect is to imagine the model composition graph with each vertex additionally labeled with the projected composed model state. Now imagine that the vertex names are shuffled according to an automorphism, while the projected composed model states remain in place. Formally,

Definition 4 *For a given composed model state, μ, and an automorphism, $\gamma \in \Gamma$, the action of γ on μ is defined as*

$$\mu^\gamma = \mu \circ \gamma,$$

where $\circ$ denotes composition of functions. For every state variable, s, in the composed model, $\mu^\gamma(s) = \mu(\gamma(s))$.

For the simple example above, Definition 4 indicates that $\mu^\gamma(B.1) = \mu(A.1)$. Having given the mathematical definition, the next step is to consider what it means in terms of the model behavior.

An automorphism of the model composition graph maps instance states among themselves. If as in the above example, the action of γ maps the state of an instance A onto the instance B, this will be denoted $B^\gamma = A$. In turn, the new state of instance A is A^γ. Determining the model behavior in the permuted composed model state requires knowledge of the set of events that are possible in the new state. Suppose $e \in E_A$ and $B^\gamma = A$. Then by the definition of Γ, which ensures

$B^\gamma = A$ if and only if A and B are instances of the same model, there must be an event, e', in B that corresponds to e in A. Since e' is to e what μ^γ is to μ, it is natural to call e' *the action of* γ *on* e and use the notation e^γ.

We will now show how Γ detects symmetry in the model. The first step is to show how Γ induces a partition of M, the set of all mappings of composed model state variables to numbers.

Definition 5 *L is a relation such that for two composed model states, μ_1, μ_2, $\mu_1 L \mu_2$ if there exists a $\gamma \in \Gamma$ such that $\mu_2 = \mu_1^\gamma$.*

Proposition 1 *L is an equivalence relation.*

Proof

L is reflexive, because a group contains an identity element. L is symmetric, since a group contains the inverse of every element. To see that L is transitive, consider μ_1, μ_2, and μ_3 such that $\mu_1 L \mu_2$ and $\mu_2 L \mu_3$. These statements imply that there exist γ_1 and γ_2 such that $\mu_2 = \mu_1^{\gamma_1}$ and $\mu_3 = \mu_2^{\gamma_2}$, so $\mu_3 = \mu_1^{\gamma_1 \gamma_2}$. Therefore L is transitive because a group is closed under multiplication. □

By Proposition 1, the automorphism group of the model composition graph partitions the state space of the composed model into equivalence classes defined by L. It will now be shown that elements in the equivalence classes of L are symmetric in the sense that all elements in a class of L have future behavior that is statistically indistinguishable. The main step in the proof is to demonstrate that for two composed model states in the same class of L, the sets of next possible states are equivalent under L.

As with the composed model state, it will be necessary to refer to projections of permuted composed model states onto instance states. The projection of μ^γ onto some instance, A, is denoted $[\mu^\gamma]_A$. It is also useful to define precisely the idea of the action of an automorphism on the local state of an instance. Given the

projection of a composed model state, μ, onto the local state of an instance, A, the action of an automorphism, γ, on μ_A must be defined. Note that this cannot be a straightforward composition of functions, since the codomain of γ is not the same as the domain of μ_A. On the other hand, it is easy to define what is meant.

Definition 6 *Let the domain of μ_A, $A \subseteq S$, be denoted $\mathcal{D}(\mu_A)$. The action of γ on μ_A is defined as*

$$[\mu_A]^\gamma = \{(s, \mu_A(\gamma(s))) \mid \gamma(s) \in \mathcal{D}(\mu_A)\}.$$

The relationship between the projection of Definition 6 and the action of an automorphism can now be explored. Suppose one delineates the local state of an instance, A, and follows it as it is moved by an automorphism to another instance, B. Now suppose one first applies the same automorphism and then examines the local state of B. In each case one sees the same local state. The formal statement is Proposition 2.

Proposition 2 *For all instance pairs (A, B) and for all $\gamma \in \Gamma$ such that $B^\gamma = A$,*

$$[\mu_A]^\gamma = [\mu^\gamma]_B.$$

Proof

Automorphisms are one-to-one and onto, so for any instance A and automorphism γ, there exists some other instance B such that $B^\gamma = A$. The set $\{s \mid \gamma(s) \in \mathcal{D}(\mu_A)\}$ is exactly the subset of composed model state variables that is projected onto the set of state variables of instance B to obtain the local state of instance B in a given composed model state. Therefore, $\mathcal{D}([\mu^\gamma]_B) = \mathcal{D}([\mu_A]^\gamma)$. This means that $[\mu_A]^\gamma$ assigns the values of the local state variables of A in composed model state μ to the corresponding local state variables of B, since $\gamma(\cdot)$ must be the same type of state variable by definition of Γ. Finally, the result follows from the definition of μ^γ. $\square$

Now that the effect of an automorphism on a composed model state has been established, the next point to consider is the state transition function in the new state, and how it relates to the state transition function of the original state. This point is the key to behavioral symmetry, since the state transition function defines what can happen next. After the relationship between state transition functions of states related by automorphism has been established, the behavioral symmetry can be characterized.

Proposition 2 is the main step in proving that states within an equivalence class of L have the same behavior. The next proposition gives the relationship between the transition functions for two states related by automorphism of the model composition graph. Informally, the proposition says that for any two states in an equivalence class of L, their sets of next possible states are related by the same automorphism as the two states themselves.

Proposition 3 *For all $\mu \in M$, $e \in E$, and $\gamma \in \Gamma$,*

$$\tau(e, \mu)^\gamma = \tau(e^\gamma, \mu^\gamma).$$

Proof

Let e be an event from an arbitrary instance called A. Then for every $\gamma \in \Gamma$ there exists an instance B such that $B^\gamma = A$. First, the automorphism γ is applied to the definition of the state transition function (Definition 3) to get

$$\tau(e, \mu)^\gamma = [(\mu - \mu_A) \cup \tau_A(e, \mu_A)]^\gamma.$$

Since $(\mu - \mu_A)$ and $\tau_A(e, \mu_A)$ are disjoint sets, the action of γ from Definition 6 can be applied to write

$$\tau(e, \mu)^\gamma = [\mu_{S-A}]^\gamma \cup [\tau_A(e, \mu_A)]^\gamma.$$

At this point, recalling that $B^\gamma = A$, it follows from Proposition 2 that

$$\tau(e, \mu)^\gamma = [\mu^\gamma]_{S-B} \cup \tau_B(e^\gamma, [\mu^\gamma]_B). \tag{2.1}$$

Finally, after rewriting (2.1) as

$$(\mu^\gamma - [\mu^\gamma]_B) \cup \tau_B(e^\gamma, [\mu^\gamma]_B),$$

and applying Definition 3, the result follows from the fact that e and A are arbitrary. □

The set of states that may be reached from a composed model state, μ, is

$$\Delta_\mu = \bigcup_{\{e \in E \,|\, \varepsilon(e,\mu)=1\}} \tau(e, \mu).$$

Each state in Δ_μ is also an element of some equivalence class with respect to L. Let H be an equivalence class with respect to L and suppose $H \cap \Delta_\mu \neq \emptyset$. In this case, H will be called a *destination class of* μ. Furthermore, when H is a destination class of μ, the set of events $\{e \in E \mid \tau(e, \mu) \in H\}$ will be denoted $E_{\mu,H}$. With these definitions we can precisely define the notion of equivalent behavior.

Proposition 4 *For all pairs of composed model states μ_1 and μ_2, if $\mu_1 L \mu_2$ then μ_1 and μ_2 have the same set of destination classes and the same transition rates to those classes.*

Proof

$\mu_1 L \mu_2$ implies there exists γ such that $\mu_2 = \mu_1^\gamma$. Therefore, the transition functions $\tau(\cdot, \mu_1)$ and $\tau(\cdot, \mu_2)$ lead to the same destination classes by Proposition 3. By the definition of Γ, there is a one-to-one correspondence, $e \leftrightarrow e^\gamma$, between the set of events that may occur in μ_1 and the set of events that may occur in $\mu_2 = \mu_1^\gamma$. Therefore, for each destination class H, $|E_{\mu_1,H}| = |E_{\mu_2,H}|$, and the total transition rate from μ_1 to H is equal to that from μ_2 to H. □

Proposition 4 establishes a localized notion of equivalent behavior. If two states are related by an automorphism of the model composition graph, then what can happen next in each state is equivalent, in the sense that each state that can be

reached from one state is related by automorphism to a state that can be reached from the other state. So Proposition 4 establishes equivalent behavior for one step into the future. To get from this proposition to a proof that the entire future behaviors of the two states are also symmetric, we use a well-known result from the theory of Markov chains, commonly known as the Strong Lumping Theorem.

Proposition 5 *The model created by replacing each equivalence class of L with a single representative state satisfies the Markov property.*

Proof

Follows directly from Proposition 4 and the Strong Lumping Theorem. □

In this section we have shown how the automorphism group of the graph may be used to detect symmetry in a model. The next section discusses the practical issues involved in exploiting the detected symmetry for the purposes of reducing the state space.

2.5 Exploiting Symmetry

As shown in Section 2.4, the automorphism group of the model composition graph may be used to detect symmetries, which can in turn be used to reduce the state space of the model. This section considers the practical issues of how to compute the automorphism group and how to use that information to directly generate a reduced state space for the composed model. The main result is a procedure for generating a compact state space for composed models.

The first step in generating a compact state space is to derive the model composition graph and compute its automorphism group. For models rich in symmetry, the order of (number of elements in) the automorphism group can be very large, but a group can be compactly described by a few of its elements, called a "generating

set." A *generating set* of a group Γ is a subset of Γ that when expanded to the smallest possible set that satisfies the properties of a group, becomes Γ. If a set S generates a group Γ, this is denoted $< S >= \Gamma$. The next problem is how to find a generating set for the automorphism group of the model composition graph.

Efficient algorithms for computing a generating set for the automorphism group of a graph have been developed in the literature on computational group theory [45, 3]. McKay [53, 54] has provided an implementation of his approach to the problem. McKay's program reads a simple graph description language and a vertex partition and produces a generating set for the automorphism group of the graph. Therefore, to find the automorphism group of the model composition graph, we translate it to the abstract format of McKay and apply his program. The result is a generating set for the automorphism group.

Given that the automorphism group is known, the next problem is how to exploit this knowledge to reduce the state space. Since detailed state spaces are very large, it is important to find a method for directly generating the reduced state space. The reduced state space contains a single state for each equivalence class, so a procedure for choosing a representative state to serve as a canonical label for each equivalence class is needed. The standard method of choosing a canonical representative of a set is to order the set and choose the minimum or maximum of the ordered elements. The model composition graph places constraints on the sorting operation. For equivalence classes of composed model states, the only permutations that may be applied to order the set are those in Γ, the automorphism group. In this case, transposing two elements means other elements may have to shift as well in order to reach a state that is still within the equivalence class. So the problem is that once a state fragment is moved to the vertex where it belongs in the canonical label, further moves must fix this state fragment at that specific vertex.

We use the concept of a "stabilizer chain" from computational group theory [11] to develop a structure-sensitive sorting procedure that solves the canonical label problem. Recall that the automorphism group, Γ, is a permutation group acting on the vertices $v \in V$. The *stabilizer of* v *in* Γ is the set $\Gamma_v = \{\gamma \in \Gamma \mid \gamma(v) = v\}$. Thus Γ_v is the set of permutations in Γ that map the vertex v to itself. The set Γ_v is a *subgroup* of Γ, which means that the elements of Γ_v are a subset of the elements in Γ, and Γ_v satisfies the properties of a group.

The idea of a stabilizer is easily extended to more than one vertex. A stabilizer $\Gamma_{v_1 v_2}$ is a subgroup of Γ_{v1} that also fixes v_2. A *stabilizer chain* is a decreasing sequence of subgroups, $\Gamma \supseteq \Gamma_{v_1} \supseteq \Gamma_{v_1 v_2} \supseteq \cdots \supseteq \Gamma_{v_1 v_2 \ldots v_k} = I$, that stabilize a growing number of vertices. As the number of stabilized vertices increases, the size of the stabilizing group shrinks. Eventually, the only stabilizing group remaining is the identity. The sequence $[v_1, v_2, \ldots, v_k]$ is called a *base* when the corresponding stabilizer chain ends in the identity. The subgroup that stabilizes the i-th component of the base will be denoted $\Gamma^{(i+1)}$, with $\Gamma^{(1)} = \Gamma$. A stabilizer chain is typically described by a base and a "strong generating set." A *strong generating set* is a set S of generators for Γ that satisfies the condition $< S \cap \Gamma^{(i)} > = \Gamma^{(i)}$.

The stabilizer chain gives us exactly what we need to find a canonical label, since it identifies the subgroups of permutations in Γ that fix vertices. A composed model state may be translated to its canonical label by maximizing the state fragment at every vertex in the base of the stabilizing chain. If $\mu_1 L \mu_2$, they must each have vertices with the same state in each subgroup $\Gamma^{(i)}$ in the stabilizer chain. Therefore, in each case, the same state will be moved to the base vertex.

The Schreier-Sims algorithm is the most efficient known deterministic algorithm for computing a base and strong generating set for a stabilizer chain of a given group. A software package that provides implementations of many algorithms from computational group theory, including Schreier-Sims, is `GAP` (Groups, Algorithms

and Programming), from RWTH Aachen [86]. Part of the GAP distribution is the GRAPE package [91], which is designed to facilitate working with graphs. GRAPE has an interface to McKay's graph automorphism program. To compute a stabilizer chain for the automorphism group of a model composition graph, we call the GRAPE package within GAP to read the generating set of the automorphism group. Then we use the GAP implementation of the Schreier-Sims algorithm to construct a stabilizer chain for the automorphism group. The next step is to exploit the stabilizer chain to produce an efficient procedure for finding the canonical label of a given composed model state.

A stabilizer chain, stored in the form of a base and a strong generating set, provides a compact description of the automorphism group. However, using the stabilizer chain to find a canonical label requires access to permutations in the subgroups that form the chain. There are several methods for generating the elements of a group from a base and strong generating set. The method used in GAP is based on a list called the "factorized inverse transversal." A *factorized inverse transversal* of the group $G^{(i)}$, containing b_i, is a list of generators, one for each point in the orbit of b_i in $G^{(i)}$, such that the k-th element in the list maps the k-th element in the orbit of b_i to some other element in the orbit of b_i that is closer (in the list of orbit elements) to b_i. By iterating this procedure and calculating the cumulative product of the permutations, one obtains the permutation that directly maps the i-th element in the orbit of b_i to b_i. Such a mapping is precisely what is needed in order to implement the procedure for producing the canonical label for a given composed model state. Figure 2.4 shows the procedure for producing a canonical label for a composed model state. The full procedure for exploiting model symmetry to generate a reduced state space is given in Figure 2.5.

We have built a prototype state generator using this approach. Following the definitions of the models, instances of the models are declared and connections

$B = [b_1, b_2, \ldots, b_n]$: Base for stabilizer chain
$O^{(i)}$: Orbit of i-th base point
K_i : Array of points in O^i with same state
for each base point $b_i \in B$
    let K_i be the array indices of the vertices in $O^{(i)}$ with the largest state
    select $\hat{k} \in K_i$ such that $\gamma_{\hat{k}}$, which moves $O^{(i)}(\hat{k})$ to b_i, leads to a
        composed model state $\mu^{\gamma_{\hat{k}}} : \mu^{\gamma_{\hat{k}}} \geq \mu^{\gamma_k}$ for all k.
    apply $\gamma^{\hat{k}}$ to move state fragment at $O^{(i)}(k)$ to b_i
end for

Figure 2.4: Procedure for canonical labeling of a composed model state

U : Unexplored states, S : Discovered states
E : Event set, E_i : Event set of instance $i \in I$
Compute automorphism group for model composition graph
Compute stabilizer chain
Initial state μ_0
Convert μ_0 to canonical label
$U = \{\mu_0\}$
$S = \{\mu_0\}$
while $U \neq \emptyset$
    choose a $\mu \in U$
    $U = U - \{\mu\}$
    $E(\mu) = \{e \in \bigcup_{i \in I} E_i \mid \varepsilon(e, \mu) = 1\}$
    for each $e \in E(\mu)$
        $\mu' = \tau(e, \mu)$
        convert μ' to canonical label
        if $\mu' \notin S$
            $S = S \cup \mu'$
            $U = U \cup \mu'$
        add arc from μ to μ' with rate $\lambda(e, \mu)$
    end for
end while

Figure 2.5: Procedure for generating compact state-space for a composed model

between places are established. By examining the specified connections, the tool constructs the model composition graph in a format suitable for McKay's algorithm and calls his program. The result is an output file containing a generating set for the automorphism group of the model composition graph. Using a `GAP` script, we process this file to compute a base and strong generating set for a stabilizer chain of the group. The script uses the factorized inverse transversal provided by `GAP` to produce a file containing the permutations that move each vertex in the orbit of a base vertex to that base vertex. Finally, this file is processed to set up the data structures for the canonical label procedure. At this point state space generation begins and is carried out according to the procedure listed in Figure 2.5.

2.6 Complexity Analysis

It is well known that the complexity of a state generation procedure is dominated by the number of states that must be generated. In general, the number of states in the model may be an exponential or combinatorial function of the number of components in the system. However, since the new procedure differs from the standard state generation procedure, it is useful to examine the impact of the changes. There are two significant differences. The new procedure computes the automorphism group of the model composition graph, and a canonical label must be generated for each state that is visited.

The computation of the automorphism group of the graph may be exponential in the worst case. However, our experience is that computing the automorphism group of the model composition graph rarely consumes more than 1 CPU second. This time is insignificant relative to the time required to generate the states. Therefore, we assume that it is unlikely to dominate the complexity of the entire procedure except in trivial cases where the state space is very small (a few hundred states, perhaps).

The canonical labeling procedure in Figure 2.4 is critical in the run time of our current prototype implementation. This procedure is called each time a state is reached. Note that a large fraction of the states in the detailed state space are not visited by our procedure, since newly discovered states that (after canonical labeling) are already in the state tree are not put in the queue of unexplored states. However, some states will be visited multiple times since they are reachable from a (possibly large) number of other states. Therefore, the number of calls to the canonical label procedure depends on the nature of the model.

We analyze the interior of the loop first. Finding the subset of vertices in the orbit of a base vertex that all share the maximal state is linear in the length of the orbit. In the worst case this is $O(v)$, where v is the number of vertices in the model composition graph. Identifying the vertex in the set of maximal vertices that will result in the maximum state when moved to the base vertex by the permutation tabled in the factorized inverse transversal is $O(vs)$, where s is the number of state variables in the composed model. Finally, applying the permutation to create the canonical label for the state is $O(s)$. Therefore, the interior of the loop is $O(vs)$.

Since we iterate over the length of the base, b, the overall asymtotic worst-case complexity of the canonical label procedure is $O(bvs)$.

The factor dominating the overall complexity of the procedure is the number of states. In the worst case, the number of states can be an exponential function of the number of vertices.

2.7 Related Work

As stated in the introduction, the problem of state-space reduction has received much attention in the literature. In this section we discuss how other research relates to our work.

The techniques presented in [12, 16] utilize colored stochastic Petri nets with restrictions so that behavioral symmetry can be derived from the definition of the color classes. These techniques use a single net with colored tokens representing different entities in the system being modeled. Our approach is not specific to stochastic Petri nets and uses a structural composition of submodels so that the system structure is retained in the model. In stochastic colored Petri nets, the model representation is compact, but the structure of the system is hidden in the definitions of the color classes and other elements of the net.

In [8], Buchholz considers a two-level hierarchical model, where from the high level point of view low level model states are distinguished only in terms of the number of entities of each type present. The entities have attributes, so they are similar to the colored tokens in [12, 16]. The models are closed, so there is no arrival of new entities. In this approach, there is no shared state, but submodels can communicate by passing entities among themselves. The low level models in [8] are specified as queueing networks, and entities pass through input and output ports defined for each second level model. Symmetry is exploited three ways. At the first level, the high-level model transition matrix is directly analyzed to discover states that are equivalent according to one of a set of permutations. At the low level, symmetry is exploited in two ways. First, identical low-level models in identical states are aggregated, and then symmetry in the behavior of different entity classes is checked. In all cases, symmetry is tested on the transition matrices, rather than at the level of the modeling formalism. While [8] indicates symmetry that may be exploited and describes a method for testing state equivalence from the transition matrix, it does not give a procedure for generating the set of permutations that describes the structural symmetry in the model. Also, the emphasis on closed systems with asynchronous communication among submodels makes this method less useful for the analysis of dependable systems.

Stochastic process algebras have been shown to be useful for exploiting behavioral symmetry in compositional models of systems containing communicating processes. In particular, the PEPA modeling formalism developed in [39] lends itself to the automatic detection and exploitation of behavioral symmetry resulting from parallel composition. Our approach uses shared state, as opposed to synchronizing processes in PEPA, to model communicating submodels.

In the reduced base model construction technique [85], a hierarchical composed model specification is built using stochastic activity network (SAN) submodels and replicate and join operations. The join operation identifies places that are common to two or more submodels. The replicate operation creates multiple copies of a designated submodel, all of which share a specified set of common places. The places that are not in the common set are distinct for each replica. Composed models are closed under the replicate and join operations, so these hierarchical models may comprise multiple levels. However, reduced base model construction can only exploit symmetry identified by the replicate operation. This limitation means, for example, that the rotational symmetry of a ring is not automatically exploited. Our new method automatically detects and exploits all symmetry in the model.

The work presented in [2] and [92] is closely related to our own and deserves special attention. In [2], the primary goal was the classification of the various configurations of failed and operational components reachable by a degradable multiprocessor system into a set of equivalence classes called "structure states." The key idea in this paper was to use a permutation group called the "symmetry group" to characterize the symmetry of the multiprocessor interconnection network. Two configurations were defined to be equivalent if one could be transformed into the other by application of a permutation in the symmetry group. The paper is devoted to the development of an algorithm for identifying each structure state and the size of the

associated equivalence class of configurations. The mathematical framework used in [2] is similar to ours, in that we also apply results from group theory. However, our work extends [2] in several ways. First, the scope of our work is much broader than the scope of [2], which was limited to multiprocessor systems consisting of hardware resources and interconnection structures that were subject to permanent faults. Our work is not limited to multiprocessor systems, but even within that class, our methods support more general fault models and repair policies. Second, we have provided a method for automatically detecting the symmetry present in a model, while the algorithm in [2] requires the modeler to specify generators for the symmetry group. Finally, our method directly constructs a fully specified Markov process that contains one state for each equivalence class, but additional manual work was required to derive a Markov process from the results of the algorithm presented in [2].

In [92], a very different approach was applied to the problem of symmetry exploitation for the purposes of reliability modeling of structured systems. This work was not based on group theory. The algorithm receives complete and explicit information on the symmetry of the system from the user. The state of the system is represented as an n-dimensional matrix where each matrix element contains the state of a component in the system. It is assumed that in each dimension there is complete freedom to permute the indices. In two dimensions, this translates to allowing all row exchanges and column exchanges. An algorithm is given for identifying equivalent configurations by computing a total order on the indices in each dimension. The algorithm first computes partial orders on each dimension by sorting the subspaces corresponding to fixing the index in one dimension. The subspaces are sorted according to the number of failed components in each subspace, and a partitioning strategy is used to obtain a total order. Our work differs from the work in [92] in several ways. First, the algorithm in [92] does not detect

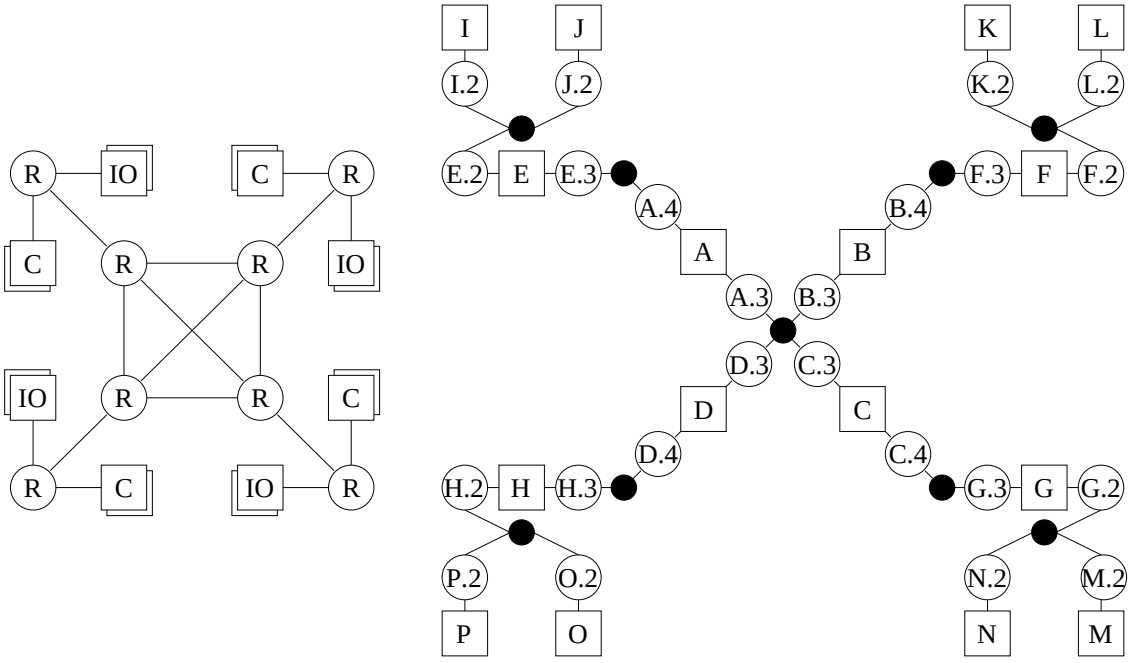


Figure 2.6: (a) Network with fully connected core and (b) model composition graph

symmetry, which must be provided as input. Second, as presented in [92], the algorithm is not suitable for more restricted symmetries, such as a ring of elements, but such symmetries are automatically detected and exploited by our methods. Third, components in [92] can only be in one of two states — operational or failed. Our methods do not impose any limitation on the component states. Finally, the state representation in [92] may need to be modified for different repair policies, but our methods work for any Markov model.

2.8 Examples and Results

In this section, we present modeling examples that demonstrate some of the symmetries that can be detected and exploited using our techniques.

Figure 2.6-a is a diagram of the first example system. This system consists

State variables

Identifier	Description	Type
r_1	router state	private
r_2	cluster state	shared
r_3	link one	shared
r_4	link two	shared

Transition function

State	Event	Next State
1,0,0,0	re_1	0,0,0,0
1,0,0,0	re_2	0,1,0,0

Event set

Identifier	Description	Enabling Condition
re_1	fails safe	$\mu(r_1) = 1$
re_2	failure propagation	$\mu(r_1) = 1$

Figure 2.7: Router model

of four clusters interconnected by a two-level point-to-point network. The core of the system is a fully connected set of four routers. Each cluster contains several processor and I/O devices. Each time a device fails, there is a chance that the failure will propagate and the operation of the whole cluster will be disrupted. For this example we do not model failure propagation between routers. The routers at the clusters can disrupt or be disrupted by a processor or I/O device, but propagation of the failure of an outside router to a core router, or *vice versa*, has not been included in this example.

The composed model for this system comprises instances of three models: one for the routers, one for the processors and one for the I/O devices. The model for the router is described in Figure 2.7. The router model has four state variables and two events. The first state variable, r_1, is private. If the router is functioning, $r_1 = 1$; otherwise it is zero. The other three state variables, r_2–r_4, are shared. The last two state variables are in fact dummy variables, which will be used to represent connections to other routers. If we choose to model failure propagation, we could use these variables, but for this model r_3 and r_4 are present in order to allow dependability measures that distinguish types of paths through the system.

The two events correspond to the two types of failure that are possible. The first is a failure that is successfully handled by the fault-tolerance mechanism of the router. The second is a failure that propagates to connected components. The coverage depends on the state of the system, so the rates for the two failure events are state-dependent, which precludes the use of fault-trees. The rate function was omitted because the specific rates for the events do not affect the size of the state space. The processor and I/O device models are the same as the router model, except that they have one link instead of two. We assume these devices must be distinguished for the dependability measure.

The composed model for the network with a fully connected core is constructed by connecting the router, processor, and I/O device models together via their shared state variables. The model composition graph is shown in Figure 2.6-b. In Figure 2.6-b, A–H are instances of the router model, I, K, M, O are processor instances, and J, L, N, P are instances of the I/O device model. The fully interconnected core is modeled by a single connection node representing the superposition of $A.3$, $B.3$, $C.3$ and $D.3$. The other routers are each connected to a core router through a superposition of link variables. For example, router E is connected to core router A via the connection node $\{A.4, E.3\}$.

This system has multiple symmetries that can be exploited. The first detected symmetry is among the four clusters extending from the core. Interchanging any two of the four core routers and their associated clusters produces an automorphism. In addition to this core symmetry, analysis of each cluster detects the symmetry among redundant processors and redundant I/O devices within a cluster. Thus, for the simple configuration of eight routers, four processors and four I/O devices, the only symmetry is the interchange of clusters, which produces twenty-four automorphisms. Adding a redundant processor to each of the four clusters results in

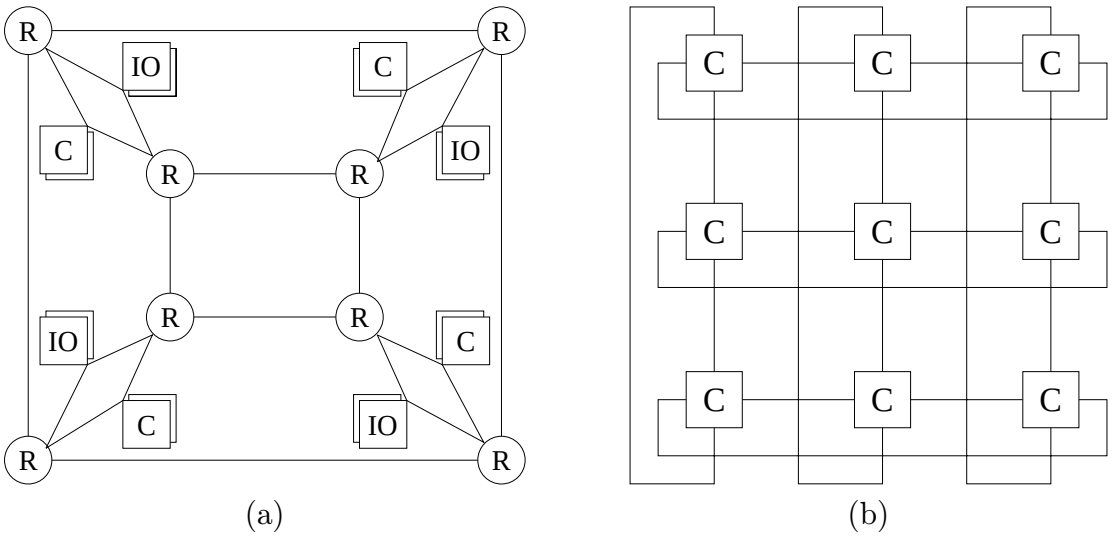
2^4 additional automorphisms. Combined with the twenty-four permutations of the four clusters, the order of the detected automorphism group grows to 384.

Table 2.1 shows the state-space compression that we achieved using the procedures presented in Sections 2.4 and 2.5. For each configuration, the column labeled "Detailed" lists the size of the state space generated by the procedure of Figure 2.3. The "Symmetry" column lists the order of the automorphism group of the model composition graph for the system, and the "Compact" column lists the size of the state space generated by detecting and exploiting symmetry. An asterisk in the column "Detailed" indicates that the state space exceeded ten million states and could not be generated within the resources available to us. Finally, when a comparison is possible, we list the relative size of the compact state space in the last column of the table. As can be seen from Table 2.1, our techniques produced reduced state spaces that were small relative to the detailed state spaces. The compression increases with the size of the state space and the order of the automorphism group.

Figure 2.8-a is a diagram of a system where the routers are configured in a double ring. Clusters are the same as in the first example, but each one of the processors and I/O devices is connected to both rings. Failure propagation within a cluster is modeled, so the failure of a component in a cluster can potentially disrupt the entire cluster. Table 2.2 shows the results for the double ring system. The symmetry for the basic system with no redundancy within the cluster consists of four rotations and an interchange of inner and outer rings, yielding an automorphism group of order eight. Adding an extra processor to each cluster introduces 2^4 processor configurations, which increases the automorphism group to 128. Finally, adding a redundant I/O device to each cluster adds 2^4 I/O configurations, bringing the order of the automorphism group to $2,048$. For the configuration using eight routers, eight CPUs and four I/O devices, the compact state space was sixteen percent of the size of the detailed state space.

Table 2.1: State space sizes for models of the fully connected system

Configuration	Detailed	Symmetry	Compact	Relative Size
6 routers, 3 CPUs, 3 I/O	19,683	6	4,599	23%
6 routers, 6 CPUs, 3 I/O	132,651	48	14,034	11%
6 routers, 6 CPUs, 6 I/O	970,299	384	43,829	4.5%
6 routers, 9 CPUs, 6 I/O	7,414,875	10,368	103,427	1.4%
6 routers, 12 CPUs, 6 I/O	*	663,552	203,282	N/A
6 routers, 9 CPUs, 9 I/O	*	279,936	239,175	N/A
6 routers, 12 CPUs, 9 I/O	*	17,915,904	472,186	N/A
8 routers, 4 CPUs, 4 I/O	531,441	24	51,734	19%
8 routers, 8 CPUs, 4 I/O	6,765,201	384	233,626	6.3%
8 routers, 8 CPUs, 8 I/O	*	6,144	1,078,975	N/A
8 routers, 12 CPUs, 8 I/O	*	497,664	3,379,453	N/A



(a) (b)

Figure 2.8: (a) Double ring and (b) toroidal mesh networks.

The final example we consider is a system configured as a two-dimensional torus, also called a toroidal mesh, shown in Figure 2.8-b. A toroidal mesh is just a mesh augmented with "wrap-around" connections. It has constant low degree and fairly good routing properties, which make it scalable and hence an attractive base for parallel computers. The toroidal mesh has also been considered for recent "network of workstations" approaches to constructing inexpensive, scalable parallel processing systems [71]. We model a toroidal mesh of workstations in which each workstation is connected to four neighbors by ethernet connections. There are no routers in this network, since the workstations use the TCP/IP protocols to do their own routing.

The model of a workstation in the torus is given in Figure 2.9. There are five state variables and three events. The first state variable, c_1, stores the operational status of the workstation. If the workstation is down because of failure propagation along a row or column, this fact is recorded in c_2 or c_3. Finally, c_4 and c_5 are used to keep track of the number of workstations in the same row and column that are down.

Although this system is highly symmetric, it is difficult to exploit with the current set of available tools. Somani [92] has considered a similar problem in his work on symmetry exploitation. Our technique automatically detects and exploits the symmetry in the toroidal mesh. The model composition graph for this system is shown in Figure 2.10. Each workstation is represented in Figure 2.10, and we also show the connections of shared variables c_2 and c_3 for each instance. The connections for c_4 and c_5 are identical, but they have been omitted from the figure to improve its clarity. The automorphism group of the model composition graph is generated by rotations of the rows and columns. For the 3x3 torus, there are 3! configurations for each row and another 3! configurations for each column, yielding an automorphism group of order 36. Likewise, the 4x4 torus has an automorphism

Table 2.2: State space sizes for models of the double ring system

Configuration	Detailed	Symmetry	Compact	Relative Size
8 routers, 4 CPUs, 4 I/O	83,521	8	19,170	23%
8 routers, 8 CPUs, 4 I/O	1,185,921	128	91,362	7.7%
8 routers, 8 CPUs, 8 I/O	*	2,048	438,082	N/A

State variables

Identifier	Description	Type
c_1	processor state	private
c_2	row crashed	shared
c_3	column crashed	shared
c_4	number down in row	shared
c_5	number down in column	shared

Transition function

State	Event	Next State
1,0,0,i,j	ce_1	0,0,0,$i+1$,$j+1$
1,0,0,i,j	ce_2	0,1,0,n,j
1,0,0,i,j	ce_3	0,0,1,i,n

Event set

Identifier	Description	Enabling Condition
ce_1	fails safe	$\mu(c_1) = 1$
ce_2	fails and disrupts the row	$\mu(c_1) = 1$
ce_3	fails and disrupts the column	$\mu(c_1) = 1$

Figure 2.9: Processor model used in toroidal mesh

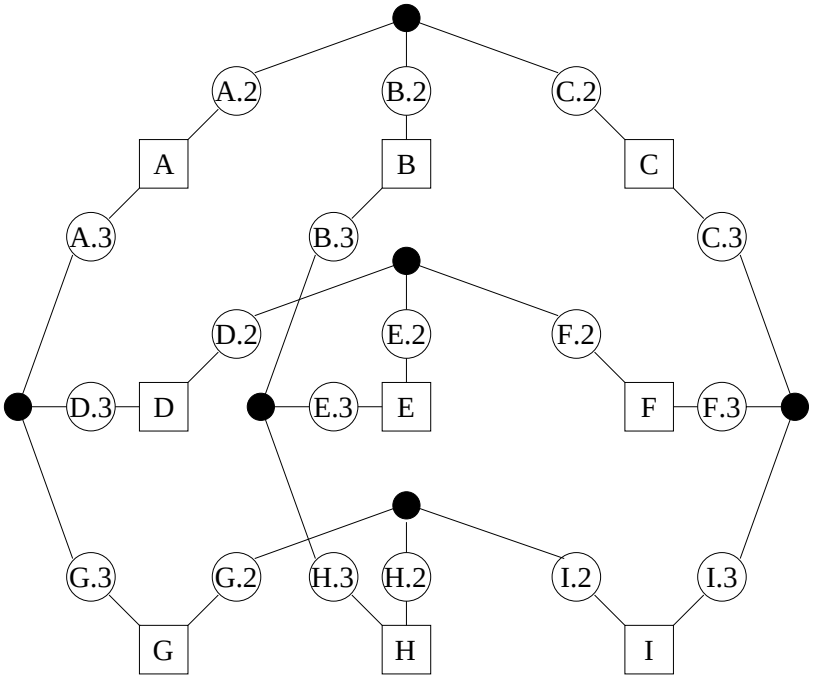


Figure 2.10: Model composition graph for the toroidal mesh system

Table 2.3: State space sizes for models of the toroidal mesh system

Configuration	Detailed	Symmetry	Compact	Relative Size
3x3 torus	22,544	36	1,109	5%
4x4 torus	9,113,820	576	39,955	0.4%
5x5 torus	*	14,400	2,118,391	N/A

group of order $4! \times 4! = 576$. Table 2.3 shows the state space sizes for three different toroidal mesh systems, ranging in size from nine to twenty-five components. As can be seen from the table, the symmetry of the torus results in a very large reduction in the number of states that must be considered. For the 4x4 torus, there is a compression of two orders of magnitude, from millions of states to tens of thousands.

2.9 Conclusion

As demonstrated in the last section, when there is symmetry in a model we can exploit it to achieve very good compression of the state space. We have presented a new approach to detecting and exploiting symmetry. In our approach, models retain the structure of the system, and all symmetry inherent in the structure of the model is detected and exploited for the purposes of state-space reduction. Many types of symmetries are detected and exploited, and the developed techniques do not require any assistance from the modeler. Results from group and graph theory are used as a rigorous foundation for the presented techniques. Specifically, we create a model composition graph from the model specification and then analyze the graph to find its automorphism group. Each model state is converted to its canonical label via a procedure using a stabilizer chain for the automorphism group, so that a reduced state space can be generated without visiting every detailed state. Using a prototype implementation of our techniques, we obtained several orders of magnitude reduction in the size of the state space for several example models.

CHAPTER 3
Path-Based Reward Variables

3.1 Introduction

Many sophisticated formalisms now exist for specifying complex system behaviors, and many tools exist that can convert a model specified in the formalism to an underlying stochastic process that can be solved. Specification methods for performance and dependability variables, on the other hand, have remained quite primitive by comparison. For example, most stochastic Petri net (SPN) tools require a user to specify performance and dependability variables in terms of a rate defined on the states of the model, and possibly, an impulse defined on each event (e.g., transition in a SPN). The trouble with this approach is that the model, as naturally defined, may not support the desired measure. To overcome this limitation, one often has to add extra components to a model (e.g. places and transitions if modeling using SPNs) to collect the desired information. These components are not part of the system being modeled, and must change whenever one desires new information from the model.

It is thus often the case that several different models of a system must be built in order to obtain the desired performance measures. Changing the model can be a time-consuming procedure, since it then must be validated to guarantee that it is still an accurate representation of the system under study. We address this problem by extending performance measure specification and state-space construction procedures to allow more flexible use of a given model. This is made possible by 1)

extending current reward variable specification methods to include variables that have "state" and can capture behavior related to sequences of events and states, and 2) extending current state-space construction algorithms that build stochastic processes that are tailored to the variable(s) of interest.

The use of performance measures to direct state-space construction is not new, but has been limited to supporting lumping based on symmetries for analytic models. In particular, [85] uses standard rate and impulse-based reward variables to put limits on the lumping that can be achieved because of symmetries in a model, and to support impulses that depend on particular activity completions. Path-based reward variables for impulses have been considered, but only to the extent that their use did not change the state space that is generated. Specifically, [78] considers the use of such variables, but limits their use to impulses on sequences of instantaneous events in order not to change the set of (stable) states that is generated.

Our work extends previous work in two important ways. First, we provide support for a more general class of reward variables for a given system model. In particular, we support the definition of measures that depend on sequences of states and events that may occur. Examples of variables whose specification is facilitated by these methods include computations of probabilities of occurrence of particular recovery actions, which have multiple steps, and computation of measures related to consecutive cell loss in ATM networks, among others. We do this by introducing the "path automaton," a finite automaton that can be used to define rewards on sequences or sets of sequences of system model states and/or transitions (both timed and untimed). By building the required memory into the performance measure specification, we simultaneously accomplish two goals: we make more flexible the specification of complex performance and dependability variables, and we avoid the need to develop multiple system models. This approach offers the advantage of a single, smaller, system model that is easier to construct and validate. Multiple

performance measures defined on multiple path automata may then be defined relative to the single system model.

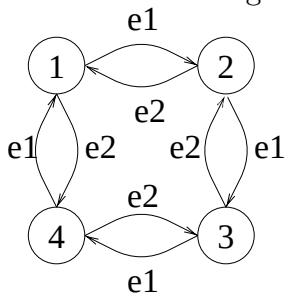
Second, we provide procedures for automatic construction of a state space that supports the specified variables from the definition of the system model, path automata, and reward structures. Note that the choice of variables as well as system model determine the state space that is generated, and different variables result in different size state spaces for the same system model. In addition, these state generation procedures include automatic support for state-space truncation for the case of performance measures defined over intervals terminated upon satisfaction of a condition on the system model, such as entrance to a particular state or the occurrence of a sequence of states and/or transitions. This is made possible by the use of path automaton "final states," which are interpreted by the state-space construction procedure as indicating that the model state reached upon entry to the final state should not be explored any further.

The remainder of the chapter is organized as follows. Section 3.3 introduces the new concept of a path-based reward variable, and Section 3.4 shows how various performance measures may be specified using path-based reward variables. Then, in Section 3.5, we present new procedures for automatically generating a state space that supports multiple path-based reward variables, and summarize the relevant numerical solution techniques. Section 3.6 gives an example model and some results on the variation of state space size for different performance measures.

3.2 Model Specification

We use the modeling formalism described in Definition 2.3. Our objective in adopting this formalism is to simplify the presentation of our ideas for variable specification and state-space construction, which are independent of the details of the actual modeling language. While this formalism is well suited to our purpose,

State transition diagram



Transition function and rate

State	Event	Next State	Rate
1	e_1	2	ϕ
	e_2	4	θ
2	e_1	3	ϕ
	e_2	1	θ
3	e_1	4	ϕ
	e_2	2	θ
4	e_1	1	ϕ
	e_2	3	θ

State variables

Identifier	Description
s_1	model state

Event set

Identifier	Description	Enabling Condition
e_1	clockwise	always enabled
e_2	counter clockwise	always enabled

Figure 3.1: Example model

it is a low-level formalism, and leads to verbose and sometimes unwieldy model definitions. However, many different high-level modeling languages can be mapped to this formalism. We demonstrate one such mapping through our use of a stochastic activity network in Section 3.6.

We now present a simple example to illustrate this modeling formalism. We will also use the same example in Section 3.5 to illustrate our state-space construction methods. Consider the state transition diagram in Figure 3.1. This system has a single state variable and two events, each of which is enabled in each state. Thus $S = \{s_1\}$ and $E = \{e_1, e_2\}$. The state variable takes values in the set $\{1, 2, 3, 4\}$, leading to state mappings $\mu_1 = (s_1, 1)$, $\mu_2 = (s_1, 2)$, $\mu_3 = (s_1, 3)$ and $\mu_4 = (s_1, 4)$. Since each event is enabled in each state, $\varepsilon(e, \mu) = 1$ for all $(e, \mu) \in E \times M$. We define the rate of occurrence of e_1 in any of the four states to be ϕ, and the rate of e_2 to be θ. The definition of the model is given in the tables of Figure 3.1.

A model executes as follows. Starting in a state, μ, the enabled events, $E(\mu) = \{e \mid \varepsilon(e, \mu) = 1\}$, compete to cause the next state transition. Each event occurs in μ at rate $\lambda(e, \mu)$. If event e occurs in μ, the next state is given by $\tau(e, \mu)$. The probability distribution over the set of states that may be reached in one transition from μ is determined in the usual way from the relative magnitudes of the event occurrence rates. In the example model of Figure 3.1, the mean holding time in each state is $\frac{1}{\phi+\theta}$ and the probability of a clockwise move via e_1 is $\frac{\phi}{\phi+\theta}$.

Models are created to answer questions about a system. We give such questions the generic name "performance measures," which we use in a broad sense, encompassing all the standard performance, dependability, and performability measures that have been defined in the literature. As pointed out by Meyer in [58], directly formulating performance measures defined at the system level in terms of a stochastic process representation of a model is not always feasible, due to the complexity of the mapping. Therefore, it is necessary (as well as desirable) to define performance measures at a high level. The concept of a "reward structure" has been established as a useful technique for specifying performance measures on models [17, 40, 79]. Defined at the model level, a reward structure associates reward accumulation rates with model states and impulse rewards with model events. Performance measures are then defined in terms of a reward structure and a variable specification, which together define a random variable called a *reward variable.* Common examples of such reward variables are the instantaneous reward at time t and the reward accumulated over the interval $[t, t + l]$.

3.3 Path-Based Reward Variables

We wish to evaluate a performance measure that is based on a sequence of model states and events. We call such a sequence a "path," and such measures "path-based performance measures." Formally,

Definition 7 *A* path, *$(\mu_1, e_1)(\mu_2, e_2), \ldots, (\mu_n, e_n)$, is a sequence of ordered pairs where μ_i is a model state and e_i is a model event.*

We say a path is *initialized* when the model enters the first state in the path. A path *completes* when the last pair in the sequence is satisfied. A path is *aborted* if the sequence is violated, for example if e_3 occurs in μ_2 instead of e_2. Definition 7 identifies a particular path, and there are many such paths in any given model. Sometimes the level of detail supported in this definition of path is not needed. For example, suppose that we are only interested in a sequence of events e_1, e_2, e_3, without regard to which states are visited. If there are many possible states that can be visited during this sequence of events, then many different paths must be specified. For situations like this, we need a convenient formalism for describing sets of paths.

To facilitate path-set specification, we introduce the "path automaton." The inputs to the automaton are a model event and the model state in which it occurred. For example, (e, μ), indicates that event e occurred in model state μ. The automaton state transition function defines the next state in terms of the current state and the input pair. Formally,

Definition 8 *A* path automaton *defined on a model, $(S, E, \varepsilon, \lambda, \tau)$, is a four-tuple, (Σ, F, X, δ), where*

- *Σ is the nonempty set of internal states;*
- *F is a (possibly empty) set of final states;*
- *$X = E \times M$ is the set of inputs; and*
- *$\delta : \Sigma \times X \rightarrow \Sigma \cup F$ is the state transition function, where for any internal state $\sigma \in \Sigma$ and input pair $x \in X$, $\delta(\sigma, x)$ identifies the next state.*

The path automaton executes as follows. Starting from an initial state $\sigma_0 \in \Sigma$, input pairs from the model are read, one for each state transition of the model. For each input pair $x \in X$, the state transition function $\delta(\sigma, x)$ identifies the next automaton state $\sigma' \in \Sigma \cup F$. If $\sigma' \in F$ then the path automaton halts. We say a path is *distinguished* by an automaton if completion of the path leaves the automaton in a final state.

We propose to evaluate path-based performance measures using path automata, together with "path-based reward structures."

Definition 9 *A* path-based reward structure *defined on path automaton* (Σ, F, X, δ) *is a pair of functions*

- $\mathcal{C} : \Sigma \times X \to \mathbb{R}$, *the* impulse reward function, *where for all internal states* $\sigma \in \Sigma$ *and input pairs* $x \in X$, $\mathcal{C}(\sigma, x)$ *is the impulse reward earned when the path automaton is in state* σ *and receives input pair* x.

- $\mathcal{R} : \Sigma \times M \to \mathbb{R}$, *the* rate reward function, *where for all internal states* $\sigma \in \Sigma$ *and model states* $\mu \in M$, $\mathcal{R}(\sigma, \mu)$ *is the rate at which reward is earned when the path automaton is in state* σ *and the model is in state* μ.

The path-based reward structure is a generalization of the standard (state-based) reward structure, since depending on the internal state of the path automaton, different rate rewards can be assigned to the same model state and different impulse rewards to the same model event. This is not possible with standard reward structures. On the other hand, any standard reward structure is easily represented using a path-based reward structure where the path automaton has only one state.

Reward variables can be constructed from the path-based reward structure and several random variables defined on the evolution of the model and the path automaton. The following random variables serve as components from which we may construct a broad array of performance measures:

- $I^t_{(\sigma,e,\mu)}$ is an indicator random variable representing the event in which at time t, the path automaton is in state σ and the last input pair was (e, μ);
- $J^{[t,t+l]}_{(\sigma,e,\mu)}$ is a random variable representing the total time during the interval $[t, t + l]$ that the automaton is in state σ and the last input pair was (e, μ); and
- $N^{[t,t+l]}_{(\sigma,e,\mu)}$ is an indicator random variable representing the number of times within the interval $[t, t + l]$ that the automaton is in state σ and receives input pair (e, μ).

Following the approach in [85], we can now define the reward variables we need for evaluating the various performance measures.

$$V_t = \sum_{(\sigma,x)\in\Sigma\times X} (\mathcal{C}(\sigma, x) + \mathcal{R}(\sigma, \mu)) \cdot I^t_{(\sigma,x)}$$

is the instantaneous reward at time t. Note that in the usual case $\mathcal{C}$ would be zero for this type of variable. Otherwise the interpretation is that the impulse associated with the most recent event that occurred prior to t is accumulated along with the rate reward corresponding to the current automaton and model state pair.

$$Y_{[t,t+l]} = \sum_{(\sigma,x)\in\Sigma\times X} \mathcal{C}(\sigma, x) \cdot N^{[t,t+l]}_{(\sigma,x)} + \mathcal{R}(\sigma, \mu) \cdot J^{[t,t+l]}_{(\sigma,\mu)}$$

is the reward accumulated over the interval $[t, t + l]$.

$$W_{[t,t+l]} = \frac{Y_{[t,t+l]}}{l}$$

is the time-averaged accumulated reward over the interval $[t, t + l]$.

So far we have considered reward variables that give us access to the value of the reward structure at fixed times or over intervals defined by fixed times. Some path-based performance measures, such as time to completion, call for an interval that is based on the random time at which some event occurs. In the literature

on stochastic processes such random times are called *stopping times.* Performance measures based on stopping times are easily handled by our formalism for path-based reward variables. We define the random variable T_F to be the instant that the path automaton enters F, the set of final states. Now we can define V_{T_F} as the instantaneous reward immediately following entry to F, and $Y_{[t,T_F]}$ as the reward accumulated from time t until entry to F.

In the next section, we show how path-based reward variables can be used to obtain various performance measures.

3.4 Example Performance Measures

Given a model and a path, there are many different questions one might ask. First, we may want to know the probability of traversing the path. Given that the path is traversed, how long does it take? How many times was the path completed in some interval? What is the chance of finding the model in the middle of traversing the path, at some arbitrary time point? What is the total time spent traversing the path within some interval? In this section we show how all of these questions may be answered using path-based reward variables.

The model described in Figure 3.1 will be used to demonstrate the use of path-based reward variables. First we identify a path. Consider the following sequence of events: starting from state 1, the process visits states 2, 3, and 4 in exactly that order, with no other excursions. The path $(\mu_1, e_1)(\mu_2, e_1)(\mu_3, e_3)$ captures this sequence of events.

To compute the probability of traversing the path before time t, we use the path automaton shown in Figure 3.2, where a path completion causes a transition to a final state.

The state transition diagram in Figure 3.2 maps to the formal tuple-notation as follows. States of the automaton appear in the diagram as labeled circles. If there

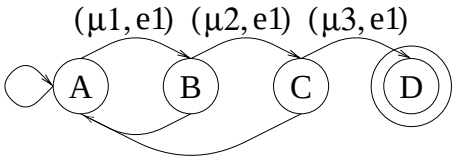


Figure 3.2: Path automaton for probability of completion of $(\mu_1, e_1)(\mu_2, e_1)(\mu_3, e_1)$

are final states, they are denoted by two concentric circles. Each arc between two states is labeled with the input that causes the transition represented by the arc. Unlabeled transitions are treated as "else" conditions. When we do not label an arc, we mean that the transition is taken if the input does not match a labeled arc.

The probability of traversing the path before time t is then the probability of finding the path automaton in its final state at time t. Thus we use the reward structure

$$\begin{aligned} \mathcal{C}(\sigma, x) &= 0 \\ \mathcal{R}(\sigma, \mu) &= \begin{cases} 1 & \text{if } \sigma = D \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{3.1}$$

and evaluate $E[V_t]$, the expected value of V_t. Since $V_t = 1$ if the path completed and $V_t = 0$ otherwise, $E[V_t]$ is the probability that the path completes before t.

To compute the time to completion of the considered path, we use the path automaton in Figure 3.2, and the reward structure

$$\begin{aligned} \mathcal{C}(\sigma, x) &= 0 \\ \mathcal{R}(\sigma, \mu) &= \begin{cases} 1 & \text{if } \sigma \neq D \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

The time to completion is the reward variable $Y_{[0,T_F]}$ where, for this automaton, $F = \{D\}$.

To count the number of completions of this path, we need to assign an impulse reward of 1 to (μ_3, e_1), but only if this event is preceded by $(\mu_1, e_1)(\mu_2, e_1)$. In addition, we do not want path completion to be a final state, since this performance

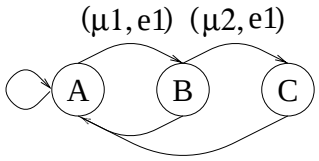


Figure 3.3: Path automaton for number of completions of $(\mu_1, e_1)(\mu_2, e_1)(\mu_3, e_1)$

measure is defined over multiple completions. The path automaton in Figure 3.3 is suitable for this performance measure. The reward structure for counting the number of completions of the path is

$$\begin{aligned} \mathcal{C}(\sigma, x) &= \begin{cases} 1 & \text{if } \sigma = C \text{ and } x = (\mu_3, e_1) \\ 0 & \text{otherwise} \end{cases} \\ \mathcal{R}(\sigma, \mu) &= 0. \end{aligned} \tag{3.2}$$

An impulse reward of 1 is assigned to the transition from path automaton state C to A if this transition is caused by event ϕ occurring in model state C. The number of completions of the path within the interval $[t, t+l]$ is $Y_{[t,t+l]}$.

The probability of finding the model on the path at time t and the time spent traversing the path within some interval are closely related; they have the same reward structure. Each performance measure can be computed by using the path automaton in Figure 3.3 and the reward structure

$$\begin{aligned} \mathcal{C}(\sigma, x) &= 0 \\ \mathcal{R}(\sigma, \mu) &= \begin{cases} 1 & \text{if } \sigma = A \text{ and } \mu = \mu_1 \\ 1 & \text{if } \sigma \in \{B, C\} \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

The expected value of V_t gives us the desired result. To get the total time spent in states on the path in the interval $[t, t+l]$ we evaluate $Y_{[t,t+l]}$.

In most cases we want to be able to obtain as many different performance measures as possible from a single state space. Our method supports multiple performance measures on a single state space, but they must be "compatible," in a way

which we will now describe precisely. A single state space can only support (one or more) fixed times, or one (random) stopping time. For this reason, if a stopping time, T_F, is used, we restrict the set of reward variables defined on a single state space such that each reward variable in the set is defined in terms of T_F. Thus two performance measures are *compatible* if they both refer to fixed times or they both refer to the same stopping time. Suppose that we define two path-based reward variables, v_1 and v_2, on a model, and that v_1 is defined on a path automaton that has a final state. For example, v_1 might measure the time to first completion of some path, p_1, while v_2 counts the number of completions of a different path, p_2. Clearly, v_1 and v_2 are most likely defined on different path automata. Yet, as stated previously, if a stopping time is present, all variables must be defined relative to that time in order to be supported by the same state space. Thus v_2 is subordinate to v_1, in the sense that v_2, if it is to be supported by the same state space as v_1, is understood to measure the number of completions of p_2 prior to the time of the first completion of p_1. For example, if we desire results for constant time t and stopping times T_{F_1} and T_{F_2}, three different state spaces are needed. Another ramification of this restriction is that a state space can only support one path automaton that has a nonempty set of final states.

This section has demonstrated how a variety of performance measures can be specified using path-based reward variables. Specific examples related to performance and dependability are given in Section 3.6. In the next section, we discuss the problem of constructing a state space that supports path-based reward variables.

3.5 State-Space Support

We have now presented path-based reward variables and demonstrated how to use them to derive various performance measures. In this section we present a method for constructing state spaces that support these variables.

The first step is to provide a precise definition of a state. We wish to allow multiple path-based reward variables to be associated with a given model. In general, this means that there will be multiple path automata and multiple reward structures to manage. Suppose that there are n different reward variables defined on a model. Each reward variable comprises a path automaton and a path-based reward structure. We index the path automaton and reward structure definitions by $i = 1, 2, \ldots, n$, so that, for example, δ_i is the state transition function for the i-th path automata. Thus we are led to the following definition of a state.

Definition 10 *For a model and a set of n path-based reward variables, a* state *is a four-tuple $(\sigma[], \mu, c[], r[])$ where*

- *$\sigma[]$ is an array of path automaton states, where $\sigma[i]$ is the internal state of the i-th path automaton;*
- *μ is the state of the model;*
- *$c[]$ is an array of impulse rewards for this state, where $c[i]$ is the impulse reward for the i-th reward variable; and*
- *$r[]$ is an array of rate rewards for this state, where $r[i]$ is the rate reward for the i-th reward variable.*

Note that $r[]$, the array of rate rewards, is determined by σ and μ, the automaton and model states, so it does not add to the state space. However, it is convenient for our presentation to include the complete reward structure in the notion of state. In an implementation, some additional flexibility can be achieved by defining rate rewards separately from the states. As stated at the end of Section 3.4, the set of path automata supported by a single state space can only include one automaton where F is nonempty. We use the convention that if there is a path automaton with $F \neq \emptyset$, it takes the first position in the array of automaton definitions.

U : unexplored states
$E(s)$: events that may occur in s;
1. Initial state $s_0 = (\sigma_0[], \mu_0, \mathbf{0}, r_0[])$
2. $U = \{s_0\}$
3. $S = \{s_0\}$
4. while $U \neq \emptyset$
5. choose $s \in U$
6. $U = U - \{s\}$
7. $E(s) = \{e \in E \mid \varepsilon(e, s.\mu) = 1\}$
8. for each $e \in E(s)$
9. $\mu' = \tau(e, s.\mu)$
10. for $i = 1$ to n
11. $\sigma'[i] = \delta_i(s.\sigma[i], s.\mu, e)$
12. $c[i] = \mathcal{C}_i(s.\sigma[i], s.\mu, e)$
13. $r[i] = \mathcal{R}_i(\sigma'[i], \mu')$
14. $s' = (\sigma'[], \mu', c[], r[])$
15. if $s' \notin S$
16. $S = S \cup \{s'\}$
17. if $\sigma'[1] \notin F$
18. $U = U \cup \{s'\}$
19. add arc from s to s' with rate $\lambda(e, s.\mu)$

Figure 3.4: Procedure for constructing a state space that supports multiple path-based reward variables

Figure 3.4 shows a procedure for constructing state spaces that support multiple path-based reward variables. In line 1, the initial state s_0 is identified by the initial state of each path automaton, the initial model state, and the initial reward structure values for each reward variable. By convention, the initial impulse rewards are zero, which is indicated by $\mathbf{0}$. In lines 2 and 3 we initialize the set of unexplored states and the set of visited states. At line 4, starting from the initial state, a breadth-first search of the state space is conducted.

First, we choose a member, s, of the set of unexplored states (line 5). The next step (line 7) is to find the set of events, $E(s)$, that may occur in the current

state. For each event the next model state, μ', is found (lines 8–9) by evaluating the model state transition function τ. At this point we have all the information needed to compute the reward structure for the new state. For each reward variable, we compute (lines 10–13) the next automaton state, the impulse reward for e occurring in μ, and the rate reward for the new automaton state, model state pair. These elements form the new state, s', constructed in line 14.

Then (line 15) we check to see if we already visited this state. If not, s' is a new state and we add it to the set of states (line 16). As long as the first path automaton in the array did not enter a final state as a result of the event that just occurred, we also add the new state to the set of unexplored states (line 18). We do not add the state to the unexplored set if it corresponds to a final automaton state, because in this case we want the state to be an absorbing state in this state space, even if it would not be an absorbing state in the model. We do this in order to support evaluation of reward variables defined in terms of stopping times. We will discuss this further at the end of this section. Finally, in line 19 we add a transition from the current state s to the new state s'. The process just described repeats until there are no remaining unexplored states.

To demonstrate the construction procedure, we use the simple example of Figure 3.1. The Markov process constructed directly from the model in Figure 3.1, without accounting for any performance measures, is shown in Figure 3.5. We now construct state spaces for two path-based performance measures, one of which is defined on an interval determined by a stopping time. The first example is the probability of path completion. To support this variable, we use the path automaton of Figure 3.2 with the reward structure in (3.1). The state space is shown in Figure 3.6.

As a second example, we consider the number of completions within an interval $[t, t+l]$. For this performance measure, we use the path automaton in Figure 3.3 and

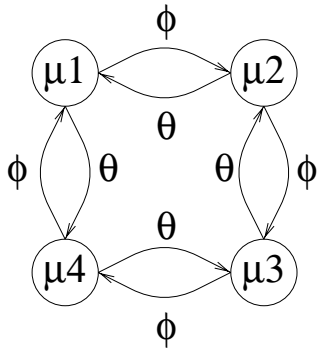


Figure 3.5: Markov process state transition diagram for simple model

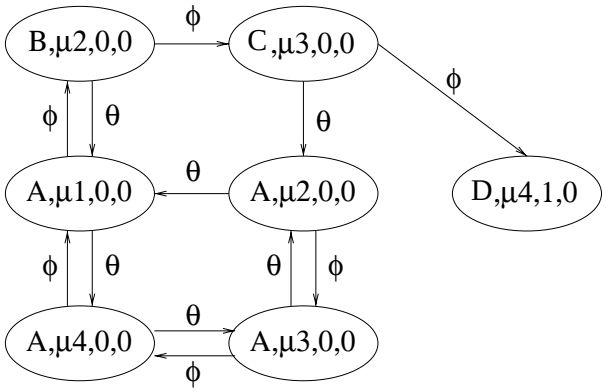


Figure 3.6: State-space supporting probability of completion of path (μ_1, e_1), (μ_2, e_1), (μ_3, e_1) by time t

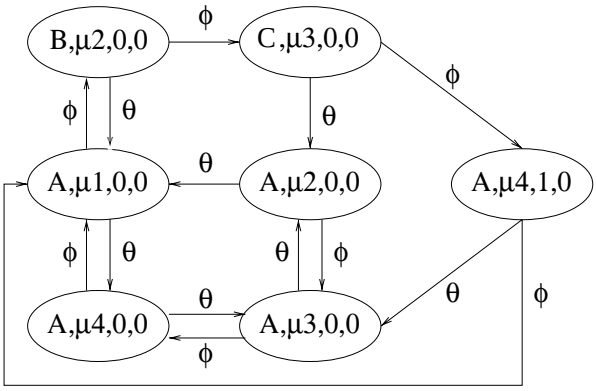


Figure 3.7: State-space supporting number of completions of path (μ_1, e_1), (μ_2, e_1), (μ_3, e_1)

Table 3.1: Methods for evaluating the reward variables

Reward Variable	Solution Method	Obtainable Information
V_t	Uniformization	Distribution
$V_{t\to\infty}$	Gauss-Seidel, SOR	Distribution
$Y_{[t,t+l]}$	Uniformization	Expected value
	Special methods (e.g. [24, 29, 62, 70, 77, 90])	Distribution
$Y_{[t,T_F]}$	Linear system (e.g. [22, 40, 64, 93])	Moments
$Y_{[t\to\infty,t+l]}$	$Y_{[0,l]}$ starting with π	Distribution

the reward structure in (3.2). The constructed state space is shown in Figure 3.7.

After the state space has been constructed, the model needs to be solved for the performance measures of interest. Table 3.1 summarizes examples of solution procedures that can be used to obtain the reward variables defined in Section 3.3. In the table, π is the vector of steady-state occupancy probabilities and SOR stands for Successive Over-Relaxation. The distribution of the instantaneous reward variable V_t is completely determined by the state occupancy probabilities at time t, which may be computed using uniformization for time t or by standard Gauss-Seidel or SOR for the limiting case $V_{t\to\infty}$. For $Y_{[t,t+l]}$, the expected value is easily obtained

using uniformization. Techniques for calculating the distribution of $Y_{[t,t+l]}$ are much more sophisticated but available (see, for example, [24, 29, 62, 70, 77, 90]).

To evaluate $Y_{[0,T_F]}$, the methods described in [22, 40, 64, 93] can be used to compute the distribution of the reward accumulated until absorption. For $Y_{[t,T_F]}$ we need the state occupancy probabilities at time t, which we then use as the initial state distribution for the methods used for $Y_{[0,T_F]}$. The time-averaged accumulated reward, $W_{[t,t+l]}$, is easily derived from Y.

3.6 Fault-Tolerant Computing Example and Results

The size of the state space that is needed to support a path-based reward variable clearly depends on the underlying model and the nature of the path automaton. In this section we introduce a larger model and investigate the variation in the size of the state space required for several path-based reward variables.

As mentioned in Section 3.2, the modeling formalism introduced there and used to develop the variable specification and state-space construction procedures is not intended to be used for large models. For the example model in this section, we use a stochastic activity network (SAN) [59] to represent the system. Each place in the SAN is a state variable in the formalism of Section 2.3. The possible stable markings of the SAN correspond to state variable mappings in the formalism. We now define the mapping between events in our formalism and events in a SAN. The completion of a timed activity, a case selection, and a sequence of instantaneous activity completions, if any instantaneous activities are enabled, is mapped to an event in our formalism. Using SAN terminology, we map each possible "stable step" [80] to an event in the formalism. The state transition function is determined by the execution rules of the SAN, as are the event-enabling function and the event rate function.

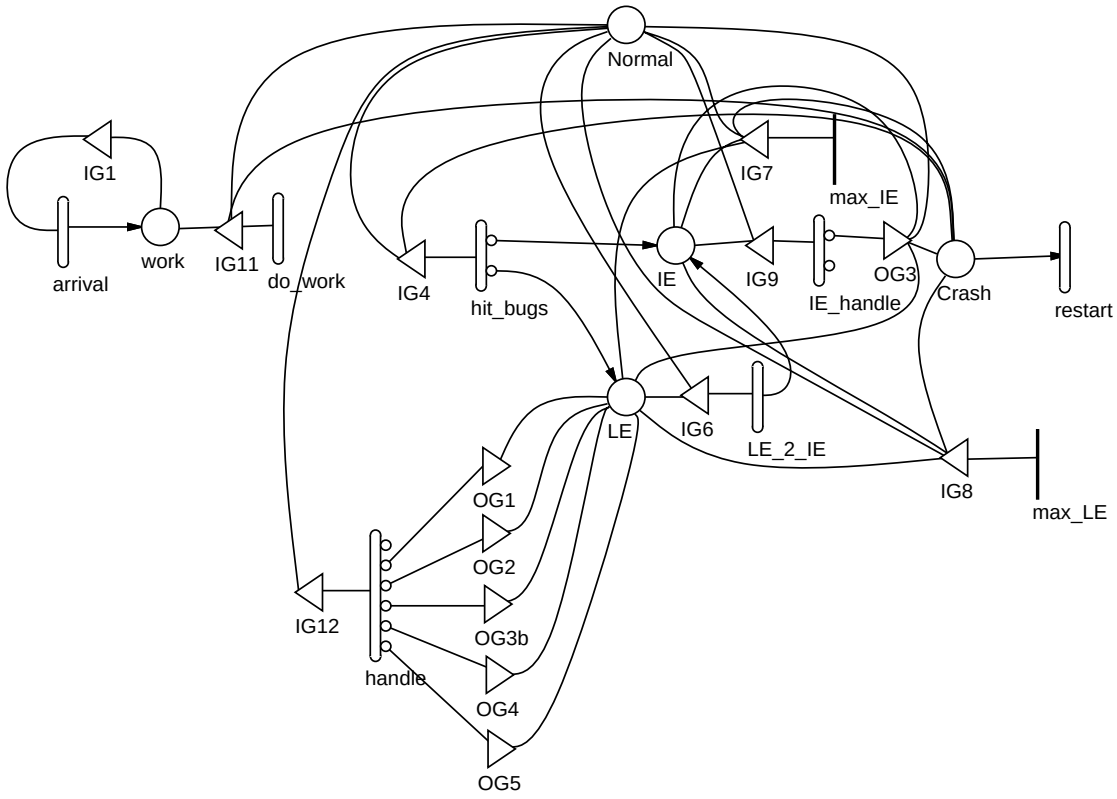


Figure 3.8: SAN model of fault-tolerant computer system

We model a computer system designed to function in the presence of software faults. A stochastic activity network model of the system is shown in Figure 3.8. The system has two modes of operation: normal and diagnostic. In normal mode, there is one token in place *Normal*, and in diagnostic mode, there are zero tokens in *Normal*. Jobs arrive to be processed according to a Poisson process defined by timed activity *arrival*. The finite buffer capacity is modeled by input gate *IG1*. When there are jobs to be processed, the system stays in normal mode. Jobs are processed at a rate defined by timed activity *do_work*. During operation in normal mode various program bugs may be encountered, at a rate governed by timed activity *hit_bugs*. The result of exercising a bug may be an immediately effective error (case 1), or a latent error. An example of an immediately effective error is an address violation; corruption of a data structure is an example of a latent error.

Immediately effective errors are handled by special error-handling routines that attempt to recover from the error. The error-handling operation is modeled by timed activity *IE_handle* and its two cases. If the error is mishandled (case 1), the system crashes. Furthermore, coincident errors can not be handled, so if an immediately effective error arrives before the last one is handled, the system crashes. This transition is modeled by instantaneous activity *max_IE* and input gate *IG7*. Latent errors do not generate any immediate problems, but the number of latent errors in the system (number of tokens in place *LE*) affects the rate at which immediately effective errors occur. This is modeled by the marking-dependent rate of timed activity *LE_2_IE*. We assume that the maximum number of latent errors that can actually be tolerated is five, so if the number of tokens in place *LE* exceeds five, input gate *IG8* enables instantaneous activity *max_LE* to model the resulting system crash.

Upon completing the available workload, the system runs a diagnostic program to check for errors. Input gate *IG11* controls the switch to diagnostic mode. When the completion of timed activity *do_work* leaves zero tokens in place *work*, *IG11* removes the token from place *Normal*, thus initiating diagnostic mode. This program is capable of detecting and removing latent errors. The latent error detection and removal process is modeled by timed activity *handle* and its six cases. The outcome of the diagnostic program depends on the number of latent errors in the system. This dependency is modeled by the six marking-dependent case probabilities defined for timed activity *handle*.

Finally, when the system crashes, it automatically begins a restart operation, which is essentially a reboot. The time it takes to restart the system is modeled by timed activity *restart*. Upon restarting, the system is as good as new.

The model parameters we used are listed in Table 3.2.

Table 3.2: Model parameters for the fault-tolerant system

Parameter	Description	Value
IE_handle rate	Rate at which bug interrupts are handled	10
LE_2_IE rate	Rate at which LE manifest as IE	0.2
MAX_IE	Maximum number of coincident IE	1
MAX_LE	Maximum number of LE that can be tolerated	5
MAX_WORK	Maximum number of jobs in the system	60
PDR (used in *handle*)	Probability of detecting and removing a LE	0.95
PHE (case 1, *IE_handle*)	Probability of handling an IE	0.95
PIE (case 1, *hit_bugs*)	Probability of IE	0.05
handle rate	Rate at which diagnostic program executes	2
arrival rate	Rate of job arrivals	1
hit_bug rate	Rate at which bugs are encountered	0.1
do_work rate	Job processing rate	2
initial_workload	System starts with one job available	1
restart rate	Rate for system restart completion	0.05

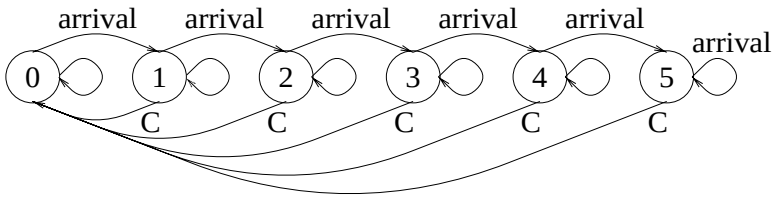


Figure 3.9: Path automaton for detecting runs of length $k \geq 5$

There are many different performance measures that can be defined for this system. Standard non-path-based measures include the distribution of the number of jobs waiting to be serviced, the availability of the system, and the fraction of time spent running diagnostics.

Several path-based performance measures are also interesting. A path automaton is well-suited to the problem of evaluating the number of times the buffer blocks as a result of a given number of consecutive job arrivals before a job completion. We refer to a sequence of k consecutive job arrivals as a *run of length* k. Figure 3.9 shows the state transition diagram of a path automaton that can be used to count the number of runs of length $k \geq 5$. Starting from automaton state zero, the au-

tomaton records arrivals by increasing its state for each arrival. If a job completion occurs, the automaton resets. If any other event occurs, the automaton remains in its current state.

Using the path automaton of Figure 3.9, we define three reward variables. The first (3.3) is simply the number of times the system blocks, which translates to the number of times a job arrives and causes the buffer to be full.

$$\begin{aligned} \mathcal{C}(\sigma, x) &= \begin{cases} 1 & \mu(work) = W_{\max} - 1 \text{ and} \\ & e = \text{arrival} \\ 0 & \text{otherwise.} \end{cases} \\ \mathcal{R}(\sigma, \mu) &= 0 \end{aligned} \tag{3.3}$$

The second (3.4) reward variable has an impulse of 1 on the event in which the automaton reaches state five, which corresponds to a run length of at least 5.

$$\begin{aligned} \mathcal{C}(\sigma, x) &= \begin{cases} 1 & \text{if } \sigma = 4 \text{ and } e = \text{arrival} \\ 0 & \text{otherwise.} \end{cases} \\ \mathcal{R}(\sigma, \mu) &= 0 \end{aligned} \tag{3.4}$$

Finally, the third variable (3.5) records an impulse of 1 each time the buffer becomes full following a run length of at least 5.

$$\begin{aligned} \mathcal{C}(\sigma, x) &= \begin{cases} 1 & \text{if } \sigma = 4 \text{ or } \sigma = 5 \text{ and} \\ & \mu(work) = W_{\max} - 1 \text{ and} \\ & e = \text{arrival} \\ 0 & \text{otherwise.} \end{cases} \\ \mathcal{R}(\sigma, \mu) &= 0 \end{aligned} \tag{3.5}$$

Blocking events are primarily of interest for performance reasons. From a dependability standpoint, we are more interested in the processes that govern the transition from normal operation to a system crash. Understanding the dominant failure mechanism in a system is important in deciding where to spend time and

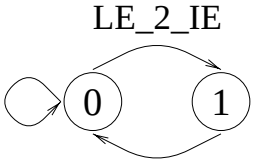


Figure 3.10: Path automaton for number of crashes caused by a latent error becoming effective and being mishandled

money to improve the dependability. An example of a path-based dependability measure is the number of crashes in an interval that result from a latent error becoming effective and then being mishandled by the error-handling routines. This event is modeled by completion of timed activity *LE_2_IE* followed by completion of *IE_handle* and selection of case 1. The path automaton that captures this sequence of events is shown in Figure 3.10. The automaton spends most of its time in state 0, but when timed activity *LE_2_IE* completes, the automaton moves to state 1. Upon the next event, the automaton returns to state 0. The reward structure

$$\begin{aligned} \mathcal{C}(\sigma, x) &= \begin{cases} 1 & \text{if } \sigma = 1 \text{ and} \\ & \text{e} = \textit{handle\_IE}, \text{ case one} \\ 0 & \text{otherwise.} \end{cases} \\ \mathcal{R}(\sigma, \mu) &= 0 \end{aligned}$$

assigns an impulse reward of 1 to the event in which *LE_2_IE* completes and is immediately followed by *IE_handle*, case 1.

Tables 3.3 and 3.4 show the sizes of the state spaces required to support the example performance and dependability measures. To see how the state space grows with increasing path length, we constructed state spaces for extended run lengths. Figure 3.11 shows the state space growth corresponding to increasing run length. The fact that the curve in Figure 3.11 levels off may, at first glance, be surprising. The explanation for this behavior is that as the run length increases there are fewer model states that may be occupied for each state in the path automaton.

Table 3.3: State space sizes for example performance and dependability measures

Description	State Space Size
Number of model states	1586
Number of crashes via the sequence LE_2_IE, IE_handle, case 1	1952
Number of runs in the workload process that are ≥ 2 and Number of overflows preceded by a run ≥ 2	3826
Number of runs in the workload process that are ≥ 3 and Number of overflows preceded by a run ≥ 3	4567
Number of overflows and Number of overflows preceded by a run ≥ 4	4567
Number of runs in the workload process that are ≥ 4 and Number of overflows preceded by a run ≥ 4	5295

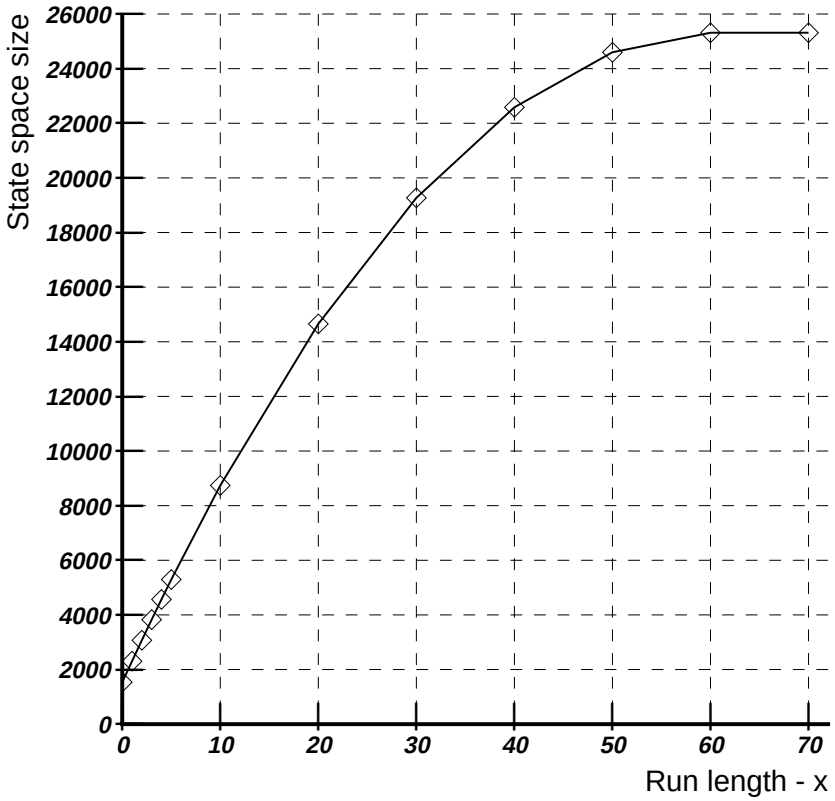


Figure 3.11: State space size versus arrival run length ($\geq$), extended to the full range of the queue

Table 3.4: Expected number of blocking events in $[0, 500]$ following runs $\geq N$

N	States	Result
Base Model	1525	N/A
N=1	2305	0.302
N=2	3072	0.199
N=3	3826	0.164
N=4	4567	0.151
N=5	5295	0.147
N=10	8740	0.138
N=20	14655	0.119
N=30	19270	0.100
N=40	22585	0.083
N=50	24600	0.069
N=60	25315	0.010
N=70	25315	0.000

For example, the model can not be on a run of length > 40 and have a queue length less than 40. Thus there are fewer additional states as the run length increases.

In Table 3.4, we have listed the results obtained for the expected number of blocking events in $[0, 500]$ that follow runs with length $\geq N$. The expected number of blocking events in $[0, 500]$ was also calculated from these state spaces, so they are larger (due to the additional impulses recorded by blocking events not caused by runs) than would have been required if we only wanted results for one variable. To validate our code, we obtained results for the SAN model using the *UltraSAN* modeling environment (see Appendix B). The results we obtained using *UltraSAN* agreed with those obtained using our code. However, we had to employ a different approach using *UltraSAN*. We added two extra places to the model and some extra code to input gates *IG1* and *IG11* in order to accumulate a count of buffer blocking events that followed runs $\geq N$. By specifying a reward structure that assigned a rate reward equal to the count of buffer blocking events, we could then solve for

the expected instantaneous rate of reward at time 500 to get the expected number of buffer blocking events in $[0, 500]$. This is computed from the transient state occupancy probabilities. To obtain a finite state space using this approach, it was necessary to limit the possible number of blocking events. We could do that for this model, since the number of such events is so low (rarely > 1). However, for longer run lengths or in situations where the number of blocking events is expected to be large, this approach will fail because the state space will grow too large.

On the other hand, with path-based reward models, the problem is mitigated by the fact that we can employ an accumulated reward solver and count impulse rewards generated by the path-based reward structure. The state space still increases with the run length, but we do not have the additional growth that would have been caused by having to accumlate the number of blocking events. The expected value of accumulated reward over an interval of time is also computable using uniformization, and the computation times are similar.

Figure 3.12 shows the graph of the data in Table 3.4. The graph has an interesting shape, with steep slopes in the initial and final stages, but a relatively flat region in between. The shape of this curve may be understood by considering the distribution of the queue length. Figure 3.13 shows the probability mass at $N = 60$, which is large due to the relatively long restart time following a crash. Once the buffer is blocked, the probability of short runs leading to additional blocking events is relatively high, which (together with the fact that all long runs finish with short runs) accounts for the initial stage of the results curve shown in Figure 3.12. But as reflected in the results, the probability decreases sharply as soon as the queue begins to empty. In the middle portion of the run length range, the curve is relatively flat, which is explained by the fact that the probability of each queue length in this range is low. The small negative slope is due to the increasing run length. Finally, Figure 3.14 shows the concentration of probability mass at the lower queue lengths.

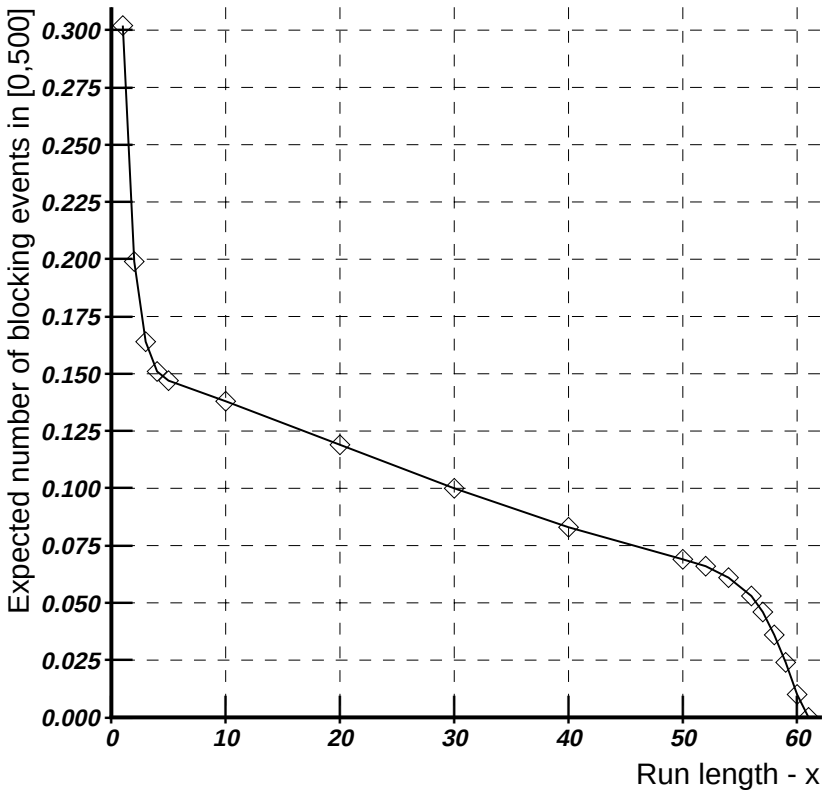


Figure 3.12: Results for expected number of runs in [0,500] that exceed N

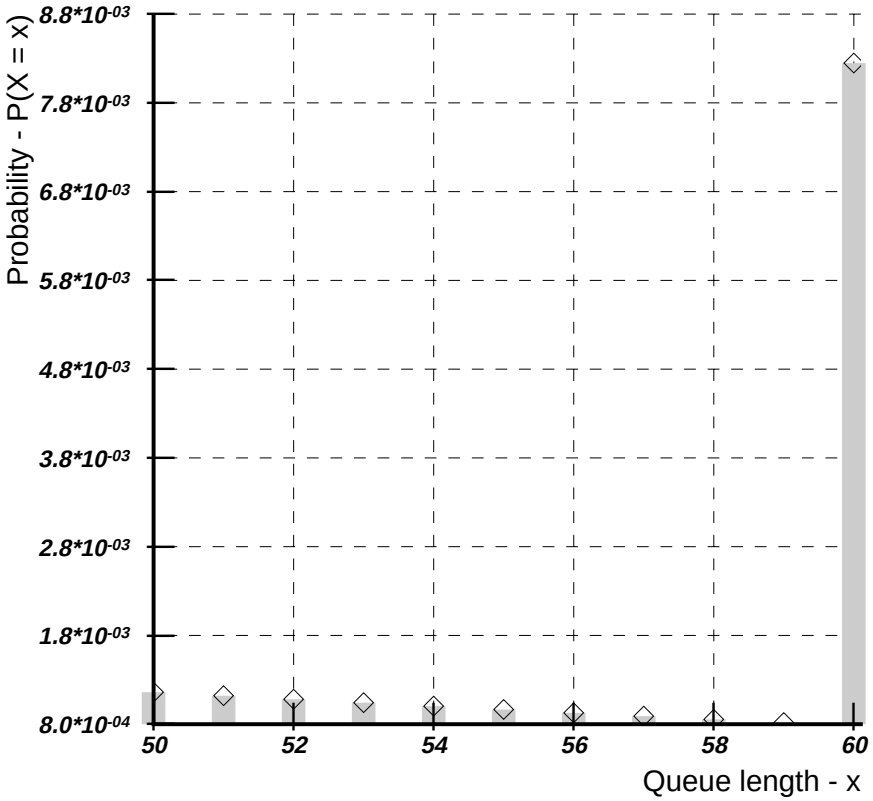


Figure 3.13: The probability mass at $N = 60$ explains the higher values and steeper slope for $N \leq 5$

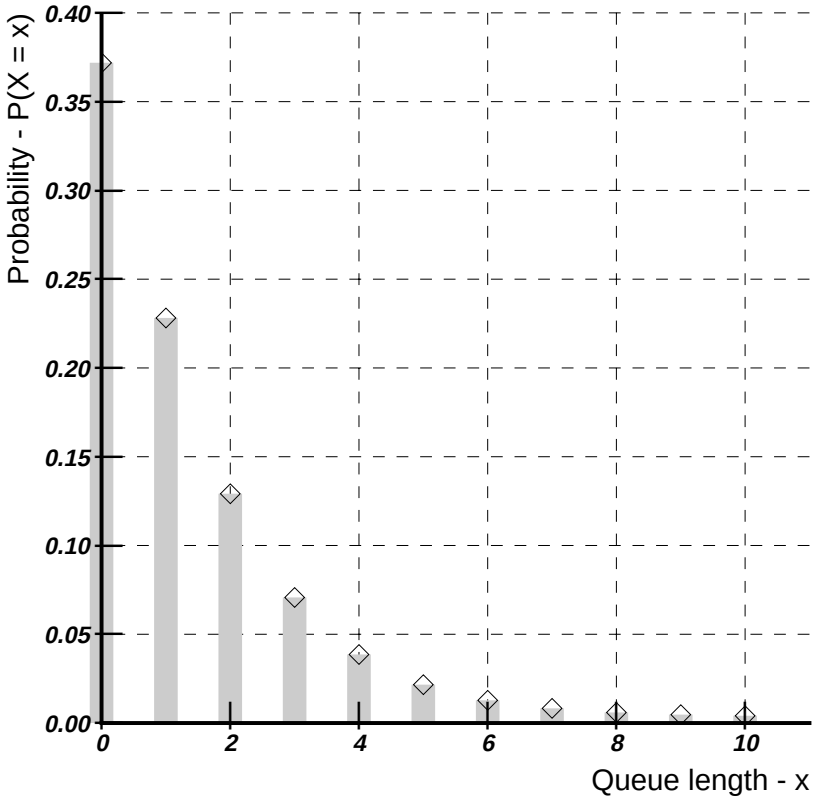


Figure 3.14: The concentration of probability mass at the lower queue length values explains the roll-off that begins at $N \geq 55$. The queue length has a mean of 4.5 and a variance of 114.

The concentration of probability mass around lower queue lengths results in a low value for the expected queue length. We computed the steady-state expected queue length as 4.5, with a standard deviation of 10.7. The steep slope in the final stage of the curve in Figure 3.12 is due to the relatively large probability masses at the lower queue lengths. To understand this, note that the probability that the queue will be in a state that admits a run of length 55 is around 0.86, but the probability of finding the queue in a state that admits a run of length 59 is only 0.6. For each step below the expected queue length, a relatively large probability mass is lost, which leads to the steep slope in the final stage of the curve.

3.7 Conclusion

This chapter presents a new performance/dependability measure specification technique and state space construction procedure that, together, solve the problem of supporting a broad array of performance measures from a given system model. Using this new approach, performance measures based on model states, model events, or sequences of (model state, model event) pairs can be supported by a single system model. Furthermore, the performance measures may be evaluated in steady state, at an instant of time, or over an interval of time defined by fixed endpoints or in terms of a (random) stopping time.

More specifically, we have shown how to specify path-based performance measures using the notion of a path automaton. With the current model state and model event as the basis for path automaton state transitions, we can use the path automaton to specify in a compact way measures that depend on particular sets of sequences of model states and events, as well as those that can be represented by standard reward variables. In addition, we have developed a state generation procedure that makes use of one or more path automata and the system model to generate a state space that is tailored to the variables of interest. In this way, a single system model can support many different performance/dependability variables, and drive the generation of many different state-level models, depending on the measure(s) of interest.

Finally, we have illustrated the use of these measure specification and state generation techniques on several small examples, to show the versatility of the approach, and on a more realistic fault-tolerant computing example, to show the variation in state-space size that can be expected for different measures. These results show the diversity of measures that can be supported by a single model.

CHAPTER 4
Symmetric Reward Variables

4.1 Introduction

As described in Definition 2 (see page 21), a composed model is made up of instances and connections. By analyzing the structure of the model composition graph, we were able to automatically detect and exploit model symmetry to reduce the state space of the composed model. The construction procedure for the reduced state space was developed under the assumption that the model structure was required and was designed by the modeler to support whatever performance measures the modeler had in mind. This is the typical state of affairs in modeling. The problem, as pointed out in our work on path-based reward variables (see Chapter 3), is that in many cases it is inconvenient to specify all the structure required for the various interesting performance measures in the system model itself. Chapter 3 thus focused on performance measures that required some history of the sequence of model states and events. By introducing and utilizing path automata to track the required sequences, and defining reward structures based on the state of the automata as well as the model, we were able to design a procedure for automatically constructing state spaces that support path-based reward variables.

The goal of this chapter is to combine the results from Chapters 2 and 3. In particular, we present a new approach for specifying performance measures that can drive a new model construction procedure for composed models that exploits symmetry. In doing so, we remove from the modeler the burden of explicitly modeling

constraints on system symmetry imposed by performance measures. This step is essential to our goal of true measure-adaptive state space construction since it is the final step required to separate the modeling of system structure and behavior from performance measurement. With this new approach, the model of the system need only reflect real system dependencies. Constraints and dependencies created by the nature of the performance measure are dealt with separately in a new reward variable formalism. Model symmetry issues are also simplified. In addition, we extend the path-based reward structures developed for single models to composed models.

4.2 Symmetric Reward Structures

In this section, we define what we mean by a symmetric reward structure and give sufficient conditions for constructing reduced state space Markov processes that support the specified reward structure.

Definition 11 *Given a composed model* (Σ, I, κ, C) *with state mapping set* M *and event set* E*, a* symmetric reward structure $(\mathcal{C}, \mathcal{R}, \Gamma_R)$ *is a pair of functions*

- $\mathcal{C} : E \to \mathbb{R}$*, the impulse reward function;*
- $\mathcal{R} : M \to \mathbb{R}$*, the rate reward function;*

and a group Γ_R *defined on the composed model such that for all* $\gamma \in \Gamma_R$, $\mathcal{C}(e^\gamma) = \mathcal{C}(e)$ *for all* $e \in E$ *and* $\mathcal{R}(\mu^\gamma) = \mathcal{R}(\mu)$ *for all* $\mu \in M$*.*

From Definition 11, it is fairly straightforward to derive a condition sufficient for correct reduced state-space construction.

Proposition 6 *Let* $(\mathcal{C}, \mathcal{R}, \Gamma_R)$ *be a symmetric reward structure defined on a composed model with automorphism group* Γ_S*. Then* $\Gamma_S \cap \Gamma_R \neq 1$*, where* 1 *indicates the trivial group, is a sufficient condition for constructing a reduced state-space Markov process that supports the reward structure.*

Proof

$\Gamma_S \cap \Gamma_R$ is a subgroup of Γ_S, which implies that every element is an automorphism of the composed model. Furthermore, the restrictions on Γ_R in Definition 11 ensure that R is invariant under all automorphisms in the subgroup, so the reward structure is supported. □

Proposition 6 shows that we can use the same procedure developed for exploiting structural symmetry in the model composition graph to handle reward structure symmetry. The proof follows from the fact that the automorphism group can only be restricted by the symmetry group of a symmetric reward structure. This means that the symmetry group ultimately used to reduce the state space is a subgroup (due to the definition of a group) of the automorphism group of the model composition graph. Therefore, the action of each permutation in the symmetry group on a given composed model state produces another composed model state that is part of the same equivalence class.

Definition 11 concisely describes a symmetric reward structure, but from a practical point of view there remain many issues that must be resolved. The main question is how to specify reward structures so that it is easy to verify the condition on Γ_R. The next section investigates the problems of specifying symmetric reward structures for composed models and deriving Γ_R.

4.3 Reward Variable Specification

In this section we introduce a reward variable formalism that will allow us to detect and exploit symmetry in the variable definition. Symmetry in the variable definition ultimately is combined with structural symmetry in the model to derive the symmetry group used to reduce the state space.

The basis for this new approach to reward variable specification is the observation that many performance measures may be written as functions that are invariant under permutations of some or all of their arguments.

Definition 12 *A function* $f(a_1, a_2, \ldots, a_n)$ *has* permutable arguments *if it is invariant under the action of a group,* Γ_f*, on its argument list. The pair* (f, Γ_f) *is called a* permutable argument function (PAF)*. The function* f *is said to be* permutable with respect to Γ_f*, which in turn will be called the* argument permutation group of f*.*

For example, consider a function of three arguments, $f(a_1, a_2, a_3)$, where for now we will assume the arguments are all elements of the same set, such as $\mathbb{R}$. Many such functions are invariant under argument permutations. Suppose f computes the average of its three arguments, for example. Then no matter how the arguments are permuted, the value of f remains the same. In this case the argument permutation group is the symmetric group $< (a_1, a_2, a_3), (a_1, a_2) >$.

We will use PAFs to define reward structures on models. The arguments of the function will be state mappings of model instances and information on the most recent event. In general, the arguments of a reward structure function will not be homogeneous. However the case of a heterogeneous argument set is easily handled by partitioning the set into cells of instances of the same model, and forming the direct product of the permutation groups defined on each cell of the partition.

Before presenting the details of the symmetry theory, we first define the new reward variable specification technique. The new technique is designed to allow relatively compact descriptions of reward structures for large composed models and to facilitate exploitation of symmetry in the reward structure. The first step in doing this is to define an "iterated reward structure."

Definition 13 *An* iterated reward function *is written* **Apply**(f, L) *where* (f, Γ_f) *is a permutable argument function and* L *is a list of argument lists for* f*. The result*

of $\mathbf{Apply}(f, L)$ *is a list,* R*, with the same number of elements as* L*. Each element* r_i *of* R *is* $f(l_i)$*, the result of* f *applied to the corresponding element of* L*.*

The iterated reward function provides a compact description of a reward function defined on a composed model. The list component, L, is a list of lists of instance state mappings. The motivation for defining reward functions this way is that composed models will often have repeated subgraphs. Rather than specify a reward function for each copy of a subgraph, the iterated reward function allows us to specify the reward function once and then specify the subgraphs to which it should be applied. Each list in L is a list of instances that comprise one subgraph. Fortunately, the list L often can be described compactly in terms of the structure of the composed model. To do this, we use the notion of the orbit of a set of instances within the automorphism group of the model composition graph.

In some cases, the performance measure implies dependencies that are not reflected in the structure of the composed model. An iterated reward function can be used to express these dependencies and, if necessary, restrict the symmetry group that is used to generate the reduced state space. In these cases, the list L is fully described and it is analyzed to generate the proper symmetry group.

The next step is to consider functions of iterated reward functions. The motivation for this additional level of complexity is that the performance measure of interest will likely be a function of the iterated reward function. For example, the performance measure might be the sum of the rewards generated from each subgraph. Other possibilities include the minimum or maximum, the number above or below a given threshold, etc. We will call a function of an iterated reward function a "compound reward function," since it is a combination of the reward functions produced by the iterated reward function.

If (g, Γ_g) is a PAF of an iterated reward function $\mathbf{Apply}(f, L)$, then Γ_g will act on the elements of L as blocks. For example, suppose f is a function of m

(f, Γ_f), (g, Γ_g) : permutable argument functions
L : list of argument lists that are compatible with f
n : number of elements in L
Γ : symmetry group

1. Construct a generating set S such that $< S >= \prod_{i=1}^{n} \Gamma_f$
2. For $\gamma \in \Gamma_g$
3. $\quad S = S \cup L^{\gamma}$
4. $\Gamma =< S >$

Figure 4.1: Procedure for constructing the symmetry group of a compound reward function

arguments, and L has n m-element argument lists. Writing out g yields

$$g(\mathbf{Apply}(f, L)) = g(f(l_{1,1}, l_{1,2}, \ldots, l_{1,m}), f(l_{2,1}, l_{2,2}, \ldots, l_{2,m}), \ldots, f(l_{n,1}, l_{n,2}, \ldots, l_{n,m})).$$

If f is a PAF with argument permutation group Γ_f, then we construct the symmetry group Γ of $g(\mathbf{Apply}(f, L))$ as follows. First we construct a set of generators for the direct product of n instances of Γ_f. Then, we add generators for the block moves of the $l_{i,j}$ according to the action of Γ_g on the arguments of g. For example, if there is a permutation in Γ_g that transposes the i-th and j-th argument of g, then we add $(l_{i,1}, l_{j,1})(l_{i,2}, l_{j,2}) \cdots (l_{i,m}, l_{j,m})$ to the generating set for Γ.

Figure 4.1 shows the procedure for constructing the symmetry group of a compound reward function. The group $< S >$ formed from the direct product of n copies of Γ_f is a group defined on the set $l_1 \cup l_2 \cup \cdots \cup l_n$. The subsets l_i are systems of imprimitivity in $< S >$, since the elements of these subsets are only permuted among themselves. In Line 3, the notation L^{γ} denotes the permutation on $l_1 \cup l_2 \cup \cdots \cup l_n$ that corresponds to permuting the subsets l_i as blocks, such that if l_i is mapped to l_j, then $l_{i,k} \rightarrow l_{j,k}$ for all k.

To construct a symmetric reward structure, we use compound reward functions to define the impulse and rate reward functions. The symmetry group of the symmetric reward structure is the intersection of the symmetry groups of the two compound reward functions. The next proposition shows that a symmetric reward structure can be specified in terms of compound reward functions.

Proposition 7 *The symmetry group constructed from a compound reward function using the procedure in Figure 4.1 satisfies the requirements placed on the symmetry group of a symmetric reward structure in Definition 11.*

Proof

Follows from the definitions of PAFs and the direct product of groups. □

4.4 Example Reward Structure Specifications

We use the composed model in Figure 4.2 to demonstrate the specification of compound reward structures. It is interesting to derive the symmetry groups for the examples, since this exercise demonstrates the power and flexibility of the representation. After each example reward structure specification, we discuss the nature of the associated symmetry group.

Example 1 Suppose we wish to evaluate the number of computers that are functioning correctly. The computers in the system are represented by instances I, K, M, and O. To specify this measure we would write $\mathcal{R} = \sum \mathbf{Apply}(f, \mathbf{Orbit}(I))$ to get the compound reward function $f(I) + f(K) + f(M) + f(O)$, where

$$f(x) = \begin{cases} 1 & \text{instance } x \text{ is working} \\ 0 & \text{otherwise.} \end{cases}$$

In Example 1, the PAF in the iterated reward function has only one argument, so it does not restrict the symmetry in any way. Furthermore, the compound reward

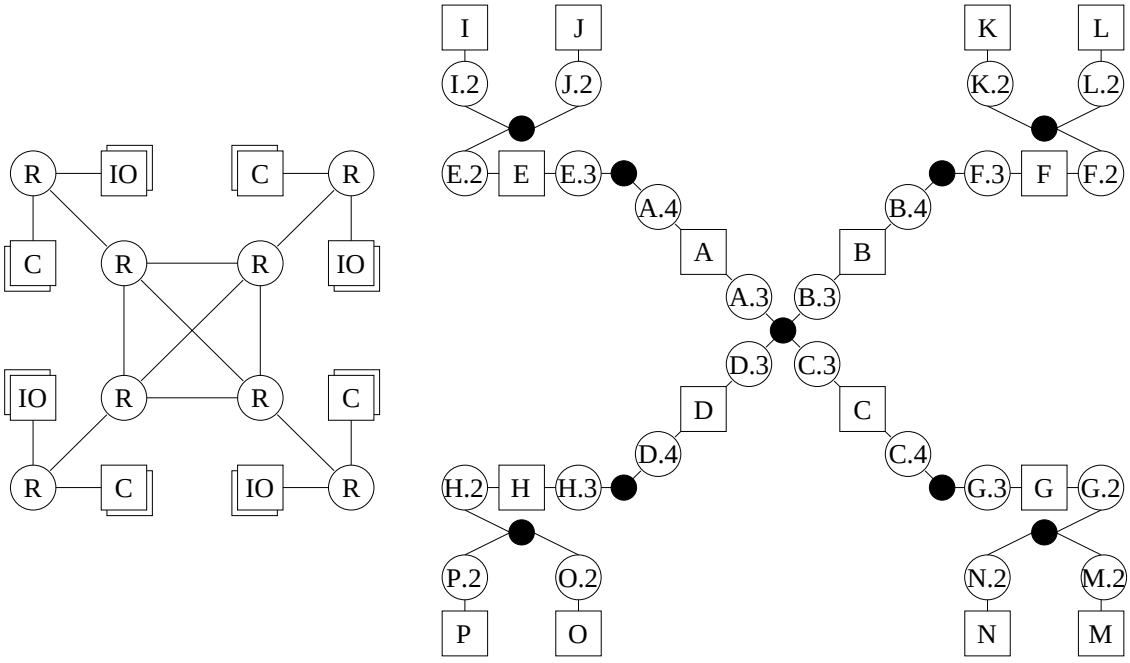


Figure 4.2: (a) Network with fully connected core and (b) model composition graph

function is constructed using addition, which is commutative, so the symmetry group is $< (I, K, M, O), (I, K) >$, which consists of all 24 permutations of the four instances. This compound reward function does not depend on any other instances in the composed model, so to form the full symmetry group of this reward function relative to the composed model, we take the direct product of this group with the symmetric group of the other instances in the model. The final symmetry group used to reduce the state space is the intersection of the full symmetry group of the compound reward function with the automorphism group of the model composition graph. In this case, the final symmetry group is simply the structural symmetry group, since the compound reward function does not place any new restrictions on that group.

Example 2 Now suppose we wanted to count the number of clusters that satisfy some property $p(a, b, c)$, where a is a computer, b is an I/O device and c is a

router on the network. (Note that p is a PAF but its argument permutation group is trivial.) Assuming p is one when the property holds and zero otherwise, in this case we would write $\sum \mathbf{Apply}(p, \mathbf{Orbit}(I, J, E))$ to get the compound reward function $p(I, J, E) + p(K, L, F) + p(M, N, G) + p(O, P, H)$. It is easy to see that this compound reward function is invariant under automorphisms of the graph, since the list of argument lists for p was generated using the automorphism group.

In Example 2, the compound reward function is more complicated, since the function operates on a subgraph comprising a router, a computer and an I/O system. These are instances of different submodels, so it makes sense that f is restricted to the identity permutation. Once again, the outer function in the compound reward function is summation, which is invariant under all permutations. In this case, the symmetry group of the reward function is the symmetric group acting on the blocks $[I, J, E]$, $[K, L, F]$, $[M, N, G]$, $[O, P, H]$, which is $< (I, K, M, O)(J, L, N, P)(E, F, G, H), (I, K)(J, L)(E, F) >$. This group is compatible with the automorphism group of the model composition graph, so the compound reward function does not restrict the symmetry group.

Example 3 Finally, consider the application of p to some subset of the clusters in the last example. For example, suppose we wish to compute

$$\sum \mathbf{Apply}(p, [(I, J, E), (M, N, G)]).$$

By restricting the list to two of the four possible clusters, we have distinguished these clusters and can no longer assume they are permutable with (K, L, F) and (O, P, H). Therefore, the symmetry group used to reduce the state space must be prevented from permuting between $\{(I, J, E), (M, N, G)\}$ and $\{(K, L, F), (O, P, H)\}$, although permutations within each set are still supported.

Example 3 demonstrates how the definition of the compound reward function can impact the symmetry group. In this case, the PAF of the iterated reward

function is the same as that in Example 2, but the list of argument lists for the PAF is different. The summation function is totally symmetric, so the symmetry group of the compound reward function is the symmetric group acting on the blocks $[I, J, E]$ and $[M, N, G]$, which is the group $< (I, M)(J, N)(E, G) >$. Forming the direct product of this group and the symmetric group over the rest of the instances, and intersecting the result with the automorphism group of the model composition graph, results in a subgroup of the automorphism group. The subgroup has only four permutations, versus the twenty-four permutations in the full automorphism group.

4.5 Symmetric Path-Based Reward Structures

In this section we extend symmetric reward structures to paths in composed models. Building on our work in Chapter 3, we define path automata for composed models and then introduce "symmetric path-based reward structures." Composed models require an extension of the theory in Chapter 3 to multiple interconnected models and automorphisms.

When a path automaton is defined on a composed model that has a nontrivial automorphism group, care must be taken in defining the state transition function. A path automaton easily can be defined on the detailed state space of a composed model, but there is a problem when we wish to use path automata in conjunction with the graph automorphism group of the model composition graph. Unless the path automaton state transition function carefully is written to be invariant under all the automorphisms of the model composition graph, the constructed state space will not in general represent a Markov process. For example, suppose that we define $\delta(\phi, e, \mu) = \phi'$. Now suppose that the canonical label for μ is μ^γ. Unless $\delta(\phi, e^\gamma, \mu^\gamma) = \phi'$, it is an error to place (e^γ, μ^γ) in the same equivalence class as (e, μ), even though both of these events are in the same Γ_S equivalence class.

To solve this problem, we introduce the "symmetric path automaton," which is compatible with the symmetry theory developed for composed models. The main difference between this new definition and that of the plain path automaton is the definition of the state transition function.

Definition 14 *A* symmetric path automaton *defined on a composed model* (Σ, I, κ, C) *with state mapping set* M *and event set* E *is a five-tuple* $(\Phi, F, X, \delta, \Gamma_P)$*, where*

- Φ *is the set of internal states;*
- F *is the set of final states;*
- $X = E \times M$ *is the set of inputs;*
- $\delta : \Phi \times X \to \Phi \cup F$ *is the state transition function; and*
- Γ_P *is a group defined on the composed model such that for all* $x \in X$ *and all* $\gamma \in \Gamma_P$, $\delta(\phi, x^\gamma) = \delta(\phi, x)$.

In Definition 14, the state transition function is defined to be invariant under the action of Γ_P on the model component of its domain. This is a subtle but important difference from Definition 8, since δ in Definition 14 can be based on representatives of the Γ_P-induced equivalence classes of X. Furthermore, since Γ_P is not required to be the same group as the automorphism group of the model composition graph, the definition of a path automaton can place additional restrictions on the symmetry that may be exploited. We will use this feature to further separate symmetry restrictions imposed by system structure from those imposed by reward variables.

For larger models, the complexity of writing a proper specification for the state transition function is too high to expect even a very patient person to go through with it. A compact language for specifying state transition functions in the presence of symmetry is needed. Fortunately, the same concepts used to develop the iterated reward structure may be applied to this problem as well. We will define the

state transition function in terms of PAFs, which will in turn define the symmetry group for the state transition function. The symmetry group of the state transition function will induce a partition of X into a set of equivalence classes that may be a refinement of the partition induced by the automorphism group.

Example 4 Consider the network model in Figure 4.2, and suppose we want to make a path automaton that transitions from ϕ_1 to ϕ_2 at the first instant that two computers are unreachable. To define this correctly, we begin by defining a PAF $(f(c, r_1, r_2), \Gamma_f)$ where

$$f(c, r_1, r_2) = \begin{cases} 1 & \text{instance } c \text{ is down} \\ 1 & \text{instance } r_1 \text{ or } r_2 \text{ is down} \\ 0 & \text{otherwise,} \end{cases}$$

and $\Gamma_f =< (r_1, r_2) >$. It is easy to see that $\sum \mathbf{Apply}(f, \mathbf{Orbit}([I, A, E]))$ indicates the number of computers that are unreachable. (Note that the orbit of the instance list $[I, A, E]$ is $[I, A, E]$, $[K, B, F]$, $[M, C, G]$, $[O, D, H]$.) To catch the condition that should trigger the transition from ϕ_1 to ϕ_2, we also need a function that identifies the right failure event. Therefore, we define another PAF $(g(c, r_1, r_2), \Gamma_g)$ where

$$g(c, r_1, r_2) = \begin{cases} 1 & \text{instances } c, r_1, r_2 \text{ are up and} \\ & \text{event } e \text{ corresponds to failure of } c, r_1, \text{ or } r_2 \\ 0 & \text{otherwise,} \end{cases}$$

and $\Gamma_g =< (r_1, r_2) >$. As defined, $\sum \mathbf{Apply}(g, \mathbf{Orbit}([I, A, E])) > 0$ indicates that a computer is about to become unreachable. Finally, we can state the condition on the transition from ϕ_1 to ϕ_2 as

$$\sum \mathbf{Apply}(g, \mathbf{Orbit}([I, A, E])) > 0 \bigcap \sum \mathbf{Apply}(f, \mathbf{Orbit}([I, A, E])) = 1. \quad (4.1)$$

The symmetry group for Condition 4.1 can be constructed using the procedure in Figure 4.1. The symmetry groups of the two components in the intersection

are the same, so we only derive the one for the sum over f applied to the orbit of $[I, A, E]$. In the first step, the generating set, S, of the direct product of four copies of Γ_f must be constructed. Since $\Gamma_f =< (r_1, r_2) >$,

$$\left(\prod_{\mathbf{Orbit}([I,A,E])} \Gamma_f \right) =< (A, E), (B, F), (C, G), (D, H) > .$$

The next step is to add generators for the block permutations of the arguments of f according to the argument permutation group of the outer function, $\sum$. As discussed before, $\sum$ is commutative, so its argument permutation group is the symmetric group over the arguments. The permutations corresponding to these block moves are generated by $(A, B, C, D)(E, F, G, H)(I, K, M, O)$ and $(A, B)(E, F)(I, K)$. Therefore, the group returned by the procedure is

$$\begin{aligned} \Gamma_P &= < (A, B, C, D)(E, F, G, H)(I, K, M, O), (A, B)(E, F)(I, K), \\ &\quad (A, E), (B, F), (C, G), (D, H) > . \end{aligned}$$

Finally, note that Γ_P does not contain any permutations the refer to the I/O instances $\{J, L, N, P\}$. In order to make Γ_P compatible with Γ_S, so that their intersection has meaning, we need to augment Γ_P by forming the direct product with the symmetric group over $\{J, L, N, P\}$.

$$\Gamma_P = \Gamma_P \times \ < (J, L, N, P), (J, L) > .$$

Now Γ_P is a group defined over the same set of instances as Γ_S. Taking the intersection $\Gamma = \Gamma_P \cap \Gamma_S$, we obtain the final symmetry group. In this example, since we used the **Orbit** operator to form the sets we end up with $\Gamma = \Gamma_S$. However, it is important to note that $\Gamma = \Gamma_S$ is a fact that was *derived*, using an unambiguous procedure, rather than assumed.

The final step in the extension of path-based reward variables to composed models is the extension of the reward structures associated with each symmetric path automaton state to symmetric reward structures.

Definition 15 *A* symmetric path-based reward structure*, defined on a symmetric path automaton* $(\Phi, F, X, \delta, \Gamma_P)$ *is a pair of functions*

- $\mathcal{C} : \Phi \times X \to I\!R$;
- $\mathcal{R} : \Phi \times M \to I\!R$;

and a group Γ_R *such that for all* $\gamma \in \Gamma_R$, $\mathcal{C}(\phi, x^\gamma) = \mathcal{C}(\phi, x)$ *and* $\mathcal{R}(\phi, \mu^\gamma) = \mathcal{R}(\phi, \mu)$.

To show that symmetric path-based reward structures are compatible with the composed models defined in Chapter 2, we need to prove that the combination can be used to construct correct Markov processes that support the specified reward variables. As for Proposition 6, the proof is straightforward if we intersect the group $\Gamma_R \cap \Gamma_P$ with Γ_S, the automorphism group of the model composition graph. Before this can be done, a procedure for constructing the group $\Gamma_R \cap \Gamma_P$ from the specification of the symmetric path-based reward structure must be developed. Once this piece is worked out, the procedure in Figure 3.4 (see page 65) can be adapted to composed models by inserting a call to the canonical label routine (Figure 2.4, page 37) for each new state that is generated.

The specification of a symmetric path-based reward structure follows naturally from the specifications of symmetric reward structures and symmetric path automata. We use PAFs to construct the state transition function for the symmetric path automaton, and we use PAFs to specify the reward structures associated with each state of the automaton. Then we augment each of these groups so that they are defined over the set of all instances and intersect them with the automorphism group of the model composition graph. The resulting subgroup is used in the state generation procedure.

Figure 4.3 shows a procedure for augmenting a group constructed from a PAF defined over some subset of the instances in the composed model. In line 1, the set D is initialized to the domain of the permutations in S, the generating set for

S : generating set for a PAF group that needs to be extended
I : instance set for composed model
Γ : new symmetry group

1. Let D be the domain of $\gamma \in S$
2. Let $A = I - D$
3. if $A = \emptyset$
4. $\quad \Gamma =< S >$
5. else
6. $\quad$ Let $\Gamma_A = \text{SymmetricGroup}(A)$
7. $\quad \Gamma =< S > \times \Gamma_A$
8. return Γ

Figure 4.3: Procedure for augmenting a symmetry group to make it compatible with the structural group

the group that is being augmented. Thus D will be the set of instances referred to by the permutations in the group generated by S. In line 2, the set A is assigned the set difference between the instance set of the composed model and the set of instances referred to by the permutations in S. If A is empty (line 3), then nothing needs to be done, since the group $< S >$ already covers all the model instances. In this case (line 4) we set the return value to $< S >$. Otherwise, in line 6, the symmetric group defined on the instances in A is constructed. Finally, in line 7 the augmented group is formed from the direct product of the group generated by S and the newly constructed symmetric group on the rest of the instances.

Figure 4.4 shows the procedure for constructing the symmetry group of a composed model with a symmetric path-based reward structure. The function Augment refers to the procedure in Figure 4.3. In the first step of the procedure, the symmetry groups are initialized to the structural symmetry group, Γ_S. Lines 2–6 form a loop iterating over the states of the path automaton. In the body of the loop, the first step (line 3) is to compute the symmetry group corresponding to conditions

$(\Phi, F, X, \delta, \Gamma_P)$: symmetric path automaton
$(\mathcal{C}, \mathcal{R}, \Gamma_R)$: reward structure on $(\Phi, F, X, \delta, \Gamma_P)$
Γ_S : structural symmetry group
Γ : new symmetry group

1. Let $\Gamma_P = \Gamma_R = \Gamma_S$
2. For each $\phi \in \Phi$
3. Compute $\Gamma_{\phi,\delta}$, the symmetry group of $\delta(\phi, \cdot)$
4. $\Gamma_P = \Gamma_P \bigcap \text{Augment}(\Gamma_{\phi,\delta})$
5. Compute $\Gamma_{R,\phi}$, the symmetry group of $(\mathcal{C}(\phi, \cdot), \mathcal{R}(\phi, \cdot))$
6. $\Gamma_R = \Gamma_R \bigcap \text{Augment}(\Gamma_{R,\phi})$
7. $\Gamma = \Gamma_S \bigcap \Gamma_P \bigcap \Gamma_R$

Figure 4.4: Procedure for constructing the symmetry group of a composed model with a symmetric path-based reward structure

for transitions out of the current state. We have named this group $\Gamma_{\phi,\delta}$ indicating it is the group corresponding to the component of δ defined for state ϕ. In Line 4, this group is augmented and intersected with Γ_P to update the symmetry group of the symmetric path automaton. Lines 5–6 do the same thing for the symmetric reward structure associated with automaton state ϕ and the reward symmetry group Γ_R. After the loop is completed, the final symmetry group is formed from the intersection of the transition function group and the reward structure group with the structural symmetry group.

4.6 Example State-Spaces for Symmetric Reward Variables

In this section, we show how the size of the state space changes according to various performance measures. The toroidal mesh shown in Figure 4.5 serves as the basis for the examples in this section. We will consider dependability measures for this system, and begin with a very simple model composition graph, shown in

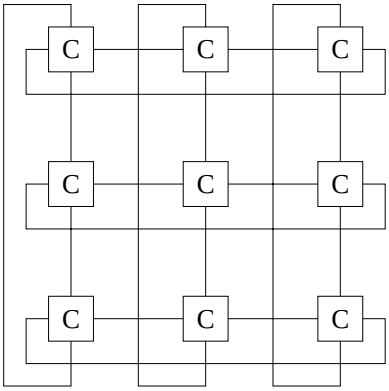


Figure 4.5: Toroidal mesh system

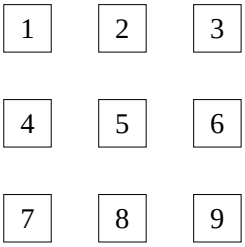


Figure 4.6: Model composition graph for toroidal mesh with independent failures

Figure 4.6. In Figure 4.6, the CPUs are independent, so the model composition graph has no arcs. The structural symmetry group is the symmetric group defined on the nine instances of the CPU model. The CPU model has only one state variable and two states: working or failed.

The first dependability measure we consider is the number of CPUs that are working at a given instant of time. We construct this measure in three steps. First, we define a function

$$g(x) = \begin{cases} 1 & \text{if } \mu_x = \text{working} \\ 0 & \text{otherwise} \end{cases}$$

on the state of a CPU model. Since we only care about the number of instances that are in the working state, we can use the iterated reward function $\mathbf{Apply}(g, \mathbf{Orbit}(1))$, which creates a list of g applied to each of the nine instances. The second step is to define the compound reward function $f(a_1, a_2, \ldots, a_9) = \sum_{i=1}^{9} g(a_i)$. Note that

Table 4.1: State-space size versus Reward Structure

Reward Structure	Number of States	Relative Size
Number of working CPUs	10	2%
Number of columns with at least one failed CPU	20	4%
Number of rows and number columns with at least one failed CPU	36	7%
Status of each CPU	512	100%

f is commutative, meaning that Γ_f is the symmetric group defined on the nine arguments. Now, for the third step, we use (f, Γ_f) as the rate reward component of a symmetric reward structure representing the desired dependability measure:

$$\begin{aligned} \mathcal{C}(x) &= 0 \\ \mathcal{R}(\mu) &= f(\mu_1, \mu_2, \ldots, \mu_9) \end{aligned}$$

Since Γ_f is already defined on all nine instances in the composed model, it need not be augmented, and Γ_R is the symmetric group over the nine instances. Therefore, the intersection $\Gamma_R \bigcap \Gamma_S = \Gamma_S$, so the full symmetric group (362,880 permutations) can be used to reduce the state space. As shown in Table 4.1, the result is a very large reduction in the number of states that must be generated. The reduced state-space is only 2% of the detailed state space.

For the second example, we consider measuring the number of columns with at least one failed CPU. In this case, although the CPUs remain independent so the structural group is the same as in the first example, the dependability measure in this example requires the CPU instances to be grouped into columns, as shown in Figure 4.7. As for the first measure, we construct this measure in three steps.

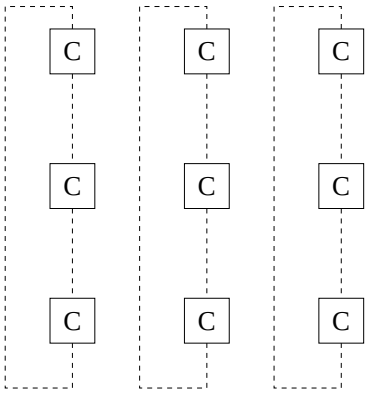


Figure 4.7: Logical grouping of CPU instances into columns

First, we define a function

$$f(a,b,c) = \begin{cases} 1 & \text{if } \mu_a = \text{failed or } \mu_b = \text{failed} \\ & \text{or } \mu_c = \text{failed} \\ 0 & \text{otherwise} \end{cases}$$

which takes the states of three CPU instances as arguments. Then we define the iterated reward function

$$\mathbf{Apply}(f, [[1,4,7],[2,5,8],[3,6,9]]),$$

which creates a list of f applied to each of the three columns. The second step is to define the compound reward function $g = \sum \mathbf{Apply}(f, [[1,4,7],[2,5,8],[3,6,9]])$. Now, for the third step, we use (g, Γ_g) as the rate reward component of a symmetric reward structure representing the desired dependability measure:

$$\begin{aligned} \mathcal{C}(x) &= 0 \\ \mathcal{R}(\mu) &= \sum \mathbf{Apply}(f, [[1,4,7],[2,5,8],[3,6,9]]) \end{aligned}$$

To derive the group Γ_R for this reward structure, we note that f and g are both PAFs that are commutative. Therefore,

$$\begin{aligned} \Gamma_{\text{cols}} = &< (1,4,7),(1,4),(2,5,8),(2,5),(3,6,9),(3,6), \\ &(1,2,3)(4,5,6)(7,8,9),(1,2)(4,5)(7,8) > . \end{aligned}$$

Since this dependability measure identifies each instance with a column, more detail is needed in the state space to support the measure. In this case

$$\Gamma_{\text{cols}} \bigcap \Gamma_S = \Gamma_{\text{cols}},$$

which with only 1296 elements is much smaller than the symmetric group over the nine instances. However, Table 4.1 shows that we still obtain a large reduction in the number of states that must be generated, compared to the 2^9 states that are required if all CPUs are distinguished.

For our third example, we further distinguish the CPUs by identifying each CPU with a row in the mesh as well as with a column. The dependability measure we are interested in is the number of rows, as well as columns, in which there is at least one failed CPU. The development of the reward structure is similar to the last measure. We define the iterated reward function $g = \mathbf{Apply}(f, [[1,2,3],[4,5,6],[7,8,9]])$. We use (g, Γ_g) as the rate reward component of a symmetric reward structure representing the dependability measure:

$$\begin{aligned} \mathcal{C}(x) &= 0 \\ \mathcal{R}(\mu) &= \sum \mathbf{Apply}(f, [[1,2,3],[4,5,6],[7,8,9]]) \end{aligned}$$

To derive the group Γ_{rows} for this reward structure, we note that g and f are both PAFs that are commutative. Therefore,

$$\begin{aligned} \Gamma_{\text{rows}} = &< (1,2,3),(1,2),(4,5,6),(4,5),(7,8,9),(7,8), \\ &(1,4,7)(2,5,8)(3,6,9),(1,4)(2,5)(3,6) > . \end{aligned}$$

Finally, since we will use this reward structure on the rows in addition to the similar structure defined for columns, we derive the final reward symmetry group by intersecting the two groups:

$$\begin{aligned} \Gamma_{\text{rows}} \bigcap \Gamma_{\text{cols}} = &< (1,2,3)(4,5,6)(7,8,9),(1,2)(4,5)(7,8), \\ &(1,4,7)(2,5,8)(3,6,9),(1,4)(2,5)(3,6) > . \end{aligned}$$

The resulting group has only 36 permutations, but as shown in Table 4.1, the reduced state-space size is still only 7% of the detailed state space size.

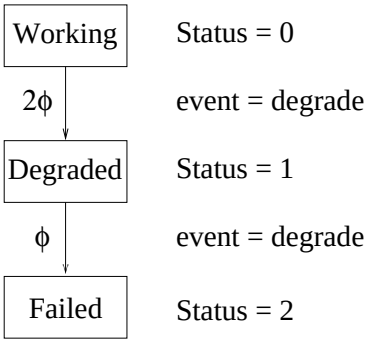
4.7 Example State-Space for Symmetric Path-Based Reward Variable

In this section we describe an example system and a symmetric path-based reward variable, and demonstrate the construction of the state-space that supports the variable. To make the presentation of the example clear, complete, and understandable, we use a small example.

Consider a cluster of two servers where each server may be in one of three states. The first state is perfect working order, the second state is partially degraded, and the last state is failed. Now suppose that this cluster is supported by an aggressive maintenance policy designed to assure a high level of availability, and consider the system available as long as at least one server is operating (even in degraded mode). Each server has its own repair facility, but repair is not completely independent. Each repair facility is aware of the state of the other server. If the system degrades to a dangerous state, where a single additional fault will bring the system to a halt, repair activity is accelerated on the server that is down. If the system does fail, repair activity is accelerated on both servers. The model for an individual server is given in Figure 4.8. This model handles the failure transitions of an individual server. Figure 4.9 shows the model of a repair facility's behavior. The repair facility maintains a single server, but its rate of work is sensitive to the state of a second server.

The model composition graph for the clustered servers is shown in Figure 4.10. Server 1 and Server 2 are instances of the server submodel, and Repair 1 and Repair 2 are instances of the repair submodel. The repair submodel has two state variables that are external. The server submodel has only one state variable (the status of

State transition diagram



Transition function and rate

State	Event	Next State	Rate
0	e_1	1	2ϕ
1	e_1	2	ϕ
2	e_1	—	ϕ

State variables

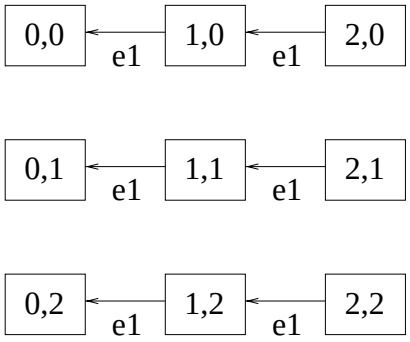
Identifier	Description
s_1	server status

Event set

Identifier	Description	Enabling Condition
e_1	degrade	$s_1 < 2$

Figure 4.8: Server model

State transition diagram



Transition function and rate

State	Event	Next State	Rate
1,0	e_1	0,0	ρ
2,0	e_1	1,0	ρ
1,1	e_1	0,1	ρ
2,1	e_1	1,1	2ρ
1,2	e_1	0,2	ρ
2,2	e_1	1,2	2ρ

State variables

Identifier	Description
s_1	server status
s_2	status of other server

Event set

Identifier	Description	Enabling Condition
e_1	repair	$s_1 > 0$

Figure 4.9: Repair model

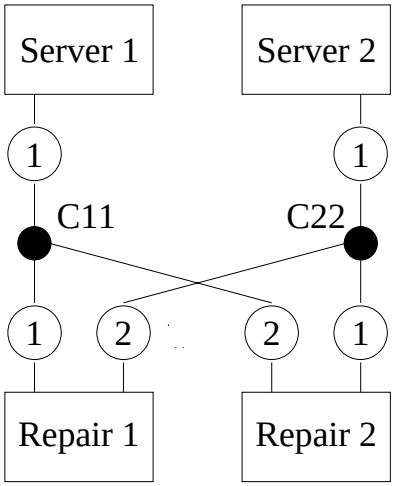


Figure 4.10: Model composition graph for clustered server

the server) and it is external. Figure 4.11-a shows the detailed state-space for the model, which contains 9 states. The accelerated repairs are reflected in the transition rates between states 12, 21, 11 and 22.

The automorphism group of the model composition graph in Figure 4.10 is obviously order two, the one permutation executing the flip of the server and repair person states. Thus for symmetric reward variables with symmetry groups that contain this permutation, the state-space can be reduced to 6 states. As shown in Figure 4.11-b, the compact state-space has 6 states, versus 9 in the detailed state-space.

As an example of a symmetric path-based reward variable, consider the expected number of times within some interval of time that starting with both servers in perfect working order, one server fails before the other degrades at all. Figure 4.12-a shows the path automaton for this path. The automaton has three states. It sits in the first state, A, until one of the servers degrades, at which point the automaton transitions to state B. If the next event is the failure of the degraded server, the automaton transitions to state C, and an impulse reward of 1 is earned. The next

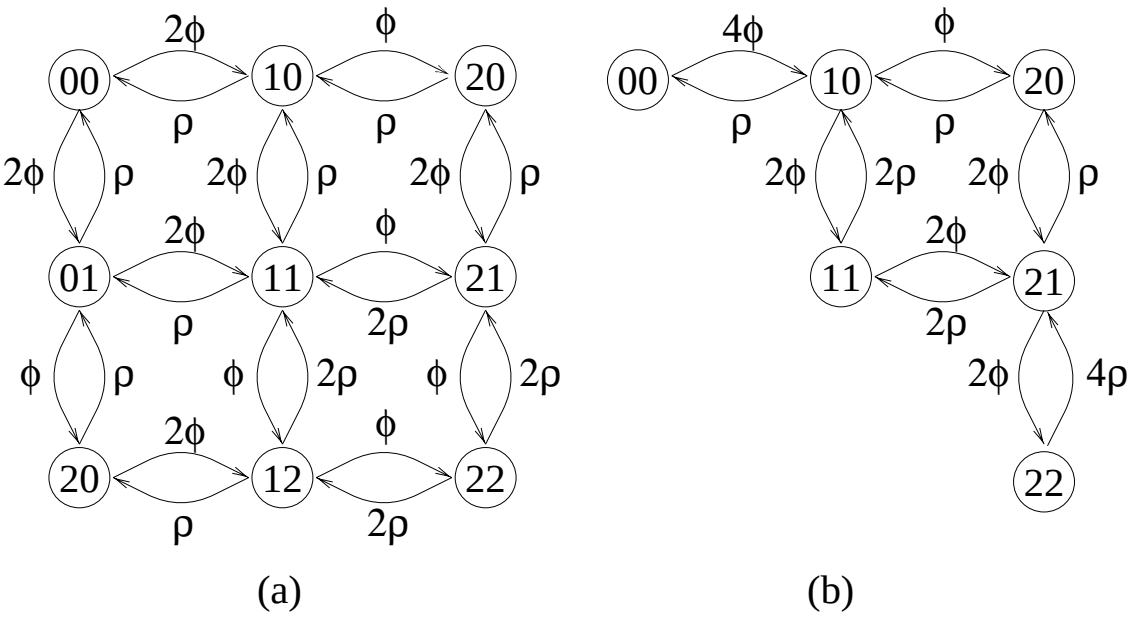


Figure 4.11: Detailed and reduced state-spaces for the example model

event returns the automaton to state A, where it stays until the system is brought back to normal working order.

The example variable is easily expressed in the formalism we have developed for symmetric path-based reward variables. The state transition function of the path automaton is expressed in terms of PAFs. The first PAF captures the event that one of the servers degrades.

$$p_{AB}(a,b) = \begin{cases} 1 & \text{if } \mu_a = 0 \text{ and } \mu_b = 0 \text{ and} \\ & e = \text{degradation of server 1 or server 2} \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

Since the AND function is commutative, the arguments of $p_{AB}(a,b)$ may be interchanged without changing the result. When applied to the cluster model, this leads to the symmetry group consisting of the interchange of the two server instances.

In state B, the transition function is based on a similar predicate, which this time matches a server status state variable mapped to 1 (degraded) with an event

for the same server that will move that server to a status of 2 (failed).

$$p_{BC}(a,b) = \begin{cases} 1 & \text{if } \mu_a = 1 \text{ and } \mu_b = 0 \text{ and } e = \text{failure of } a \text{ OR} \\ & \mu_a = 0 \text{ and } \mu_b = 1 \text{ and } e = \text{failure of } b \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

As written in Equation 4.3, p_{BC} is a simple logical OR between the condition tested on each server's status state variable. Since logical OR is commutative, the function has a symmetry group of order 2 that corresponds to the interchange of the two server instances.

To obtain the symmetry group of the symmetric path-based reward variable defined using p_{AB} and p_{BC}, we use the procedure listed in Figure 4.4. The symmetry groups of states A and B are the same — they contain the identity and the interchange of the two server instances. The reward structure is very simple, consisting of an impulse reward of 1 earned upon transition to state C. From state C the automaton transitions to state A on the next event, without regard to model state or event, so the state transition function for C is completely symmetric. Therefore, at the end of the loop defined by lines 2–6 of the procedure in Figure 4.4, Γ_P is generated by the permutation that interchanges the two servers and the permutation that interchanges the two repair people. Then, in line 7, the intersection with the automorphism group of the model composition graph reduces the symmetry to the permutation that simultaneously interchanges the two server states and the two repair person states.

By using the symmetric path-based reward variable, we are able to exploit the symmetry in the model so that the path-based variable does not expand the state-space nearly as much as it would otherwise. Figure 4.12-b shows the reduced state space that supports the path-based reward variable. The reduced state-space is one state smaller than the original detailed state-space. By exploiting the symmetry in this example, we have removed the cost associated with retaining the

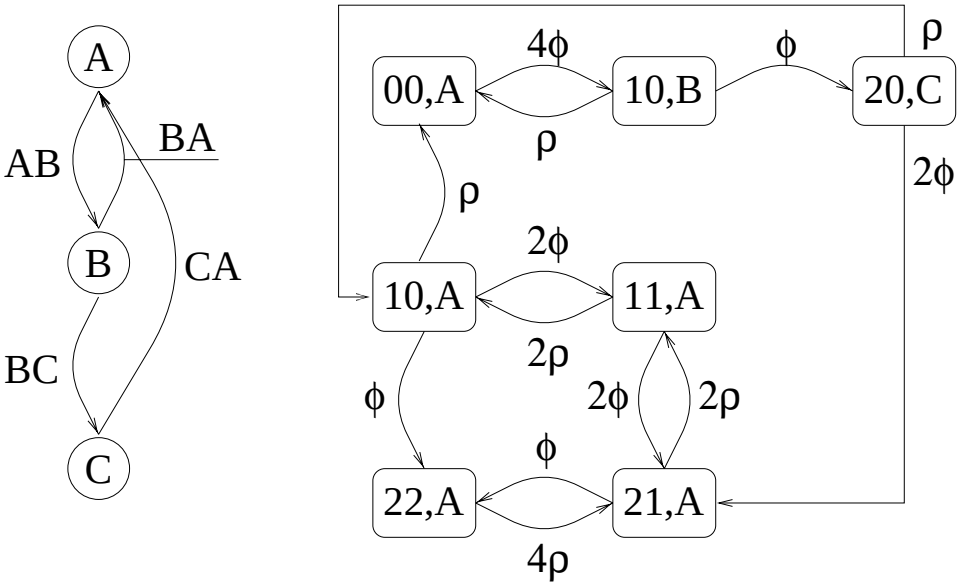


Figure 4.12: Path automaton and extended state-space

history needed to support the path-based reward variable. This example serves to demonstrate the advantages of exploiting symmetry whenever possible.

4.8 Conclusion

We have presented new techniques for specifying performance, dependability and performability measures and automatically constructing state-spaces tailored to model and measure symmetry. This new theory for state-space construction will allow us to construct tools that are easier to use and that produce smaller state-spaces that can be solved in less time than the state-spaces produced by the current state-of-the-art tools.

CHAPTER 5
Conclusion

This dissertation has addressed the problem of automatically constructing state spaces that are tailored to model and measure symmetries. The three main contributions are 1) new methods for detecting and exploiting model symmetry, 2) development of path-based reward variables, and 3) a new theory of symmetry detection and exploitation that combines the symmetries of the measures and the model structure to adapt the amount of state-space reduction. The result of this work is a set of procedures for specifying models and reward variables that remove many of the traditional headaches of modeling, and a set of state-space construction techniques that allow the modeler to achieve large reductions in model state-spaces with much less effort than is required by traditional methods.

APPENDIX A Definitions and Results from Group Theory

In this appendix we present the definitions and results from the theory of groups that are used in our work. One of the standard references for group theory is Hall [51]. The more elementary but very nicely written book by Mathewson [52] is a gentle and lucid introduction to the basic concepts of group theory.

A *group* is a collection of elements together with a product operation that satisfies the following properties.

Closure For every pair of elements in the group, the product exists and is a unique element of the group.

Associative law If a, b and c are elements of the group, then $(ab)c = a(bc)$;

Identity There is an element, I, in the group such that $Ia = aI = a$.

Inverse For every element a in the group, there is another element, a^{-1}, in the group, called the inverse of a, such that $aa^{-1} = a^{-1}a = I$.

The commutative property is not required for a group. The *order* of a group is the number of elements in the group.

A permutation is a one-to-one mapping of a set onto itself. There are several common methods for representing a permutation. The straightforward representation is a two line quantity that explicitly defines the mapping of each element onto another element of the set. For example, consider the set $\{1, 2, 3\}$. A permutation of this set that swaps 1 and 2 is represented in two-line notation as

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix}.$$

A shorthand notation keeps just the bottom line, which would lead to 213. However, even this is redundant, since 3 is not moved but it is still represented. For long permutations, this redundancy becomes tedious. Therefore, the preferred notation for permutations is the so-called cycle notation. The permutation that swaps 1 and

2 is simply represented as (12). The interpretation is that of a circular list, where each element is mapped to the one following it in the list. Thus (12) maps 1 to 2, and then wraps around so that 2 is mapped to 1.

A product of permutations is the composition of the mappings. For example,

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix}.$$

In cycle notation, $(12)(132) = (23)$.

Permutation groups are one of the most useful types of group. A permutation group is a group whose elements are permutations of some set. The *degree* of a permutation group is the number elements permuted by elements of the group. For example, the set of all permutations of $1, 2, 3$ form a group called the symmetric group of degree three. The elements of the group expressed in one-line notation are $\{123, 132, 213, 231, 312, 321\}$. The product operator in a permutation group is composition of the permutation maps.

The *automorphism group of a graph* is a permutation group on the set of vertices, V, of the graph. Each permutation in the automorphism group relabels vertices in such a way as to maintain adjacency. That is, if two vertices are joined by an edge in the initial graph labeling, then they also will be joined by an edge after a permutation in the automorphism group has been used to relabel the vertices.

Groups can be very large, so rather than describing a group by enumerating all its elements, it is preferable to use a more compact description. One such description is a set of "generators" for the group. A *generating set* for a group is a subset of the group elements which combine to produce all the elements in the group. For example, the group consisting of all possible permutations of $\{1, 2, 3, 4\}$ (called the symmetric group of degree four) is generated by $(1, 2, 3, 4)$ and $(1, 2)$. In this way, a group with sixteen elements is compactly represented by two. To refer to the group Γ generated by a set E of elements, we use the notation $\Gamma =< E >$.

The *direct product* of two groups A and B is denoted $A \times B$, and is constructed using the product rule

$$(a_1, b_1)(a_2, b_2) = (a_1 a_2, b_1 b_2).$$

A permutation group is *imprimitive* if the set of elements upon which the permutations act may be divided into disjoint sets $S_1, S_2, \ldots, S_n$, such that every element of Γ either maps the elements of S_i onto themselves or onto the elements of another set S_j. If a group is imprimitive the sets S_i are called *systems of imprimitivity*.

APPENDIX B *UltraSAN*

THE *UltraSAN* MODELING ENVIRONMENT[1]

B.1 Abstract

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

[1]This appendix consists of the article "The *UltraSAN* Modeling Environment," co-authored with W. H. Sanders, M. A. Qureshi and F. K. Widjanarko, and published in Performance Evaluation, vol. 24, pp. 89–115, 1995.

paper illustrates the use of the package in the solution for the unreliability of a fault-tolerant multiprocessor using two solution techniques.

B.2 Introduction

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

Due to these merits and the computational complexity of most model solution methods, the development of effective software tool support is an active research area. Many such tools exist, and are based on a wide variety of model specification techniques (e.g., queueing networks, Markov processes, object oriented languages, and stochastic Petri nets). Several surveys have been written on such tools, see, for example, [34, 36, 94]. Stochastic Petri nets and extensions have been the subject of significant recent research resulting in a number of software tools supporting the evaluation of systems represented in this way. These tools include among many DSPNexpress [48], FIGARO [7], GreatSPN [15], METASAN [84], RDPS [30], SPNP [18], SURF-2 [5], TimeNET [35], and *UltraSAN* [23]. Some of these tools have been built to illustrate particular classes of solution algorithms (e.g., DSPNexpress,

analytic solution of SPNs with deterministic and exponential timings), while others have had very broad goals, implementing both analytic- and simulation-based solution methods (e.g., GreatSPN, SPNP, SURF-2, and *UltraSAN*).

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

Since the initial release in 1991 [23], many new construction and solution techniques and user interface features have been added. In particular with respect to the user interface, a parameterized model specification method that facilitates solution of models with multiple sets of input parameter values has been developed. Furthermore, three new analytical solvers, as well as a terminating simulation solver based on importance sampling, have been added to the package. The first new analytic solver solves models that have deterministic as well as exponential activities. The two remaining new analytic solvers compute the probability distribution function and mean, respectively, of reward accumulated over a finite interval. The importance sampling simulator provides an efficient solution for models that contain rare events which are significant relative to a measure in question. These enhancements have improved *UltraSAN*'s usefulness to others, and illustrated its effectiveness as a

test-bed for new modeling techniques. The package has now been at academic and industrial sites for about four years, and much has been learned from its use.

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

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

B.3 Overview

The organization of *UltraSAN* follows directly from the goals of the package. As shown in Figure B.1, the package is constructed in a modular manner. All of the modules shown are coordinated by a central program called the control panel, that manages the interface between the user and the various modules, and provides basic consistency checking. In addition, the control panel offers several useful utilities, such as automatic documentation and convenient viewing of output data. The modules communicate via files which have well-defined formats. This

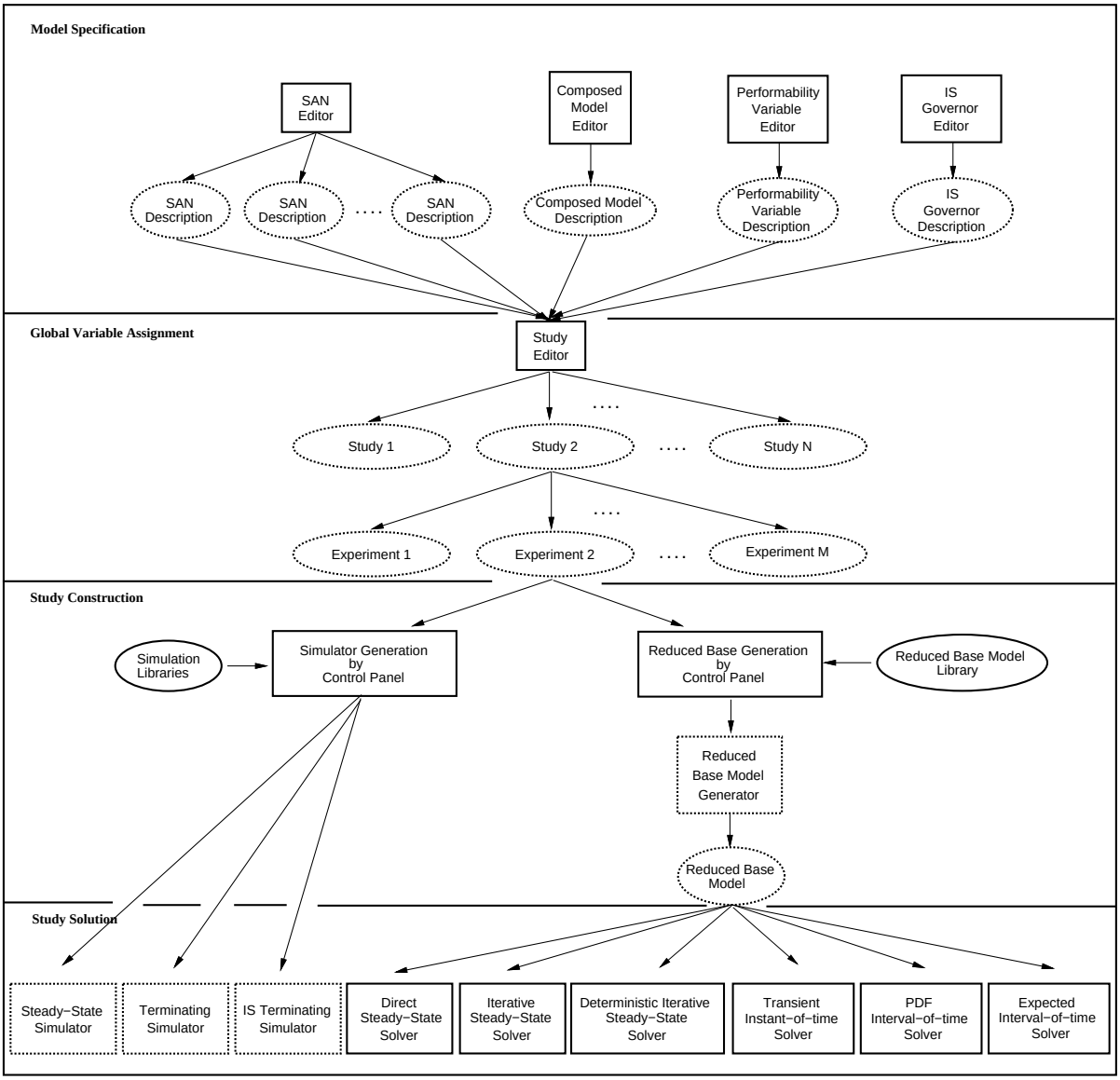


Figure B.1: Organization of *UltraSAN*.

configuration makes it easy to add new functionality, thus contributing to the goal of being a test-bed for developing new modeling techniques.

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

Model specification

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

Global variable assignment

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

Study construction

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

Analytical models are constructed by automatically generating a Markov process representation of the model. The size of the generated Markov process is determined in part by the definition of state. In standard stochastic Petri net models, the standard definition of the state space is the set of tangible reachable markings. Realistic models of modern systems usually have very large state spaces. This problem has been widely recognized, and various approaches have been devised to handle the explosive growth [89, 94]. *UltraSAN* uses an innovative largeness-avoidance approach called reduced base model construction [85]. This technique uses information about the performance measures and symmetries in the model to directly generate a stochastic process that is often much smaller than the set of reachable markings, and yet is sufficient to support the specified performance measures. Experience has shown that reduced base model construction is effective for medium to large systems that have replicated subsystems [83].

When the developed model is intended to be solved using simulation, *UltraSAN* does not generate a state-level representation. For this solution method, study construction is accomplished by linking the model description files generated in model specification and global variable assignment to the simulation library. The result is an executable simulation of the model. Essentially, simulation is accomplished by assigning events to activity completions and using the rules of SAN execution to determine the next state. Data structures designed for reduced base model construction [81] make simulation more efficient. Symmetries in the model that allow lumping in state space construction also help reduce the amount of future events list management necessary for each state change.

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

Study solution

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

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

PDF interval-of-time solver uses uniformization to determine the distribution of accumulated reward over a specified interval of time. Similarly, the expected interval-of-time solver uses uniformization to compute the expected value of interval-of-time and time-averaged interval-of-time variables.

Three solution modules are available for simulation. The steady-state simulator uses the method of batch means to estimate the mean and variance of steady-state measures. The terminating simulator uses the method of independent replications to estimate the mean and variance of instant-of-time, interval-of-time, and time-averaged interval-of-time measures. The third simulator is useful when the model contains events that are rare, but significant with respect to the chosen performance variables. In this case, traditional simulation will be inefficient due to the rarity with which an event of interest occurs. For problems like this, *UltraSAN* includes an importance sampling simulation module [68] that estimates the mean of a performance measure. The importance sampling simulation module is flexible enough to handle many different heuristics, including balanced failure biasing, and is also applicable to non-Markovian models [67].

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

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

B.4 Model Specification

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

Input to each of the four editors is graphical and X-window based. Detailed instructions concerning the operation of each editor cannot be given due to space limitations, but can be found in [14]. However, a brief description of what must be specified for each step follows.

SAN model specification

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

The functions of the three model primitives are as follows. *Places* are as in Petri nets. Places hold tokens, and the number of tokens in each place is the *marking* of the SAN. The interpretation of the number of tokens in a place is left to the discretion of the modeler. Sometimes tokens are used to represent physical quantities such as the number of customers in a queue, or the number of operational

components in a system. On the other hand, it is often convenient to use the marking of a place to represent the state of a network, or the type of a customer.

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

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

Gates serve to connect places and activities, and are an important part of the modeling convenience offered by SANs. *Input* gates specify when activities are enabled, and what happens to the input places when the activity completes. The predicate of an input gate is a boolean function of its input places, which when true causes the activity to be enabled. Each input gate also has a function that specifies how the marking of the input places is updated when the activity completes. In most stochastic Petri nets (e.g., GSPNs), enabling conditions and marking updates are specified using networks of immediate transitions and places. Input gates allow functional descriptions of the enabling predicate and marking updates, which is more convenient for specifying complicated enabling conditions and token movements that are common in models of real systems. *Output* gates allow functional

specifications of how the marking of output places is updated upon completion of an activity. This flexibility is enhanced when output gates are used in combination with cases. Introduced first in SANs in [61], the utility of gates has become apparent, and similar features have been incorporated into other extensions to stochastic Petri nets, such as the guards in SRNs [17].

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

The fault-tolerant multicomputer system discussed in [44] serves as a good example to illustrate the modeling process in *UltraSAN.* Although this system is a pure dependability model, it incorporates some interesting features that challenge a model description language. As in [44], a multicomputer consists of multiple computer modules connected via a bus, where each module is as shown in Figure B.2. Each computer module consists of three memory modules, three CPUs, two I/O ports, and an error handler. Each computer module is operational if at least two memory modules, two CPUs, one I/O port and the error handler are operational. The multicomputer is considered to be operational if at least one of the computer

modules is operational. As given in [44], reliability is modeled at the chip level, and each chip has a failure rate of 1×10^{-7} failures per hour or 8.766×10^{-4} failures per year.

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

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

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

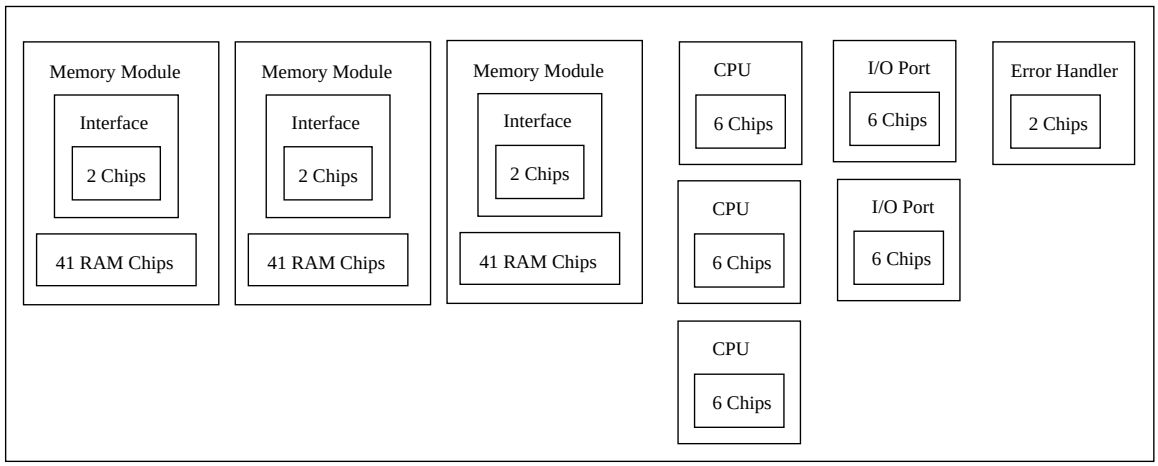


Figure B.2: Block diagram of fault-tolerant computer

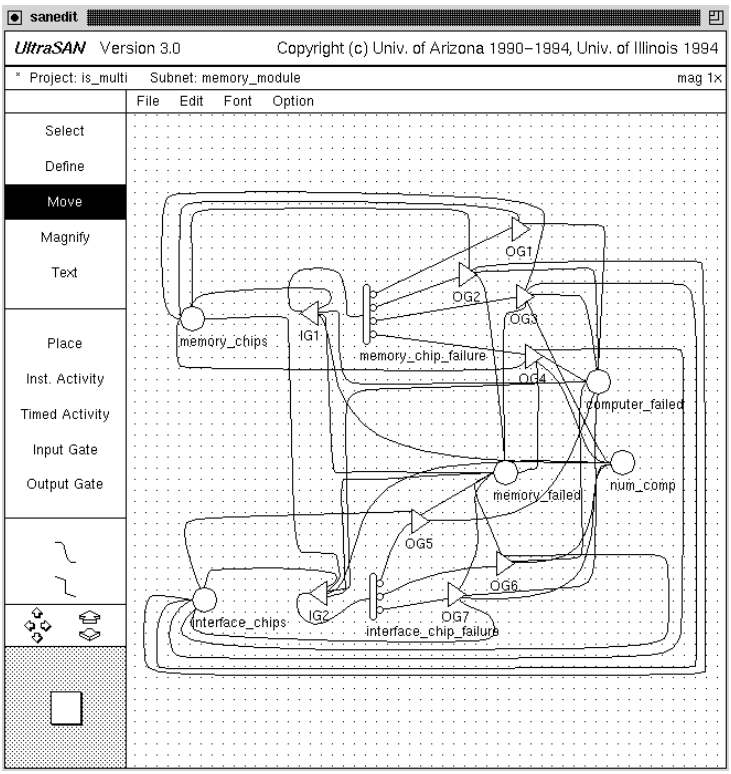


Figure B.3: SAN model of a memory module, as specified in the SAN editor.

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

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

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

Table B.1: Coverage Probabilities in Multicomputer System

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

Table B.2: Example Input Gate Definitions

Gate	*Definition*
IG1	Predicate $(MARK(memory\_chips) > 38)$ && $(MARK(computer\_failed) < 2)$ && $(MARK(memory\_failed) < 2)$
	Function *identity*
IG2	Predicate $(MARK(interface\_chips) > 1)$ && $(MARK(memory\_failed) < 2)$ && $(MARK(computer\_failed) < 2)$
	Function $MARK(memory\_chips) = 0;$

Table B.3: Example Output Gate Definition

Gate	*Definition*
OG2	$MARK(memory\_chips) = 0;$ $MARK(interface\_chips) = 0;$ $MARK(memory\_failed)++;$ *if* $(MARK(memory\_failed) > 1)$ $\quad MARK(computer\_failed)++;$

effort to reduce the state space. Finally, the last case corresponds to a RAM chip failure that is not covered at any level, causing the entire multicomputer to fail.

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

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

The activity time distributions, case distributions, and gate predicates and functions are all defined using auxiliary editors that pop up over the SAN editor. The

Table B.4: Activity Case Probabilities for SAN Model *memory_module*

Activity	*Case*	*Probability*
interface_chip_failure	1	*0.95*
	2	*0.0475*
	3	*0.0025*

memory_chip_failure	1	*if* (*MARK*(*memory_chips*) == *39*)
		  return(*0.0*);
		else
		  return(*0.998*);

	2	*if* (*MARK*(*memory_chips*) == *39*)
		  return(*0.95*);
		else
		  return(*0.0019*);

	3	*if* (*MARK*(*memory_chips*) == *39*)
		  return(*0.0475*);
		else
		  return(*0.000095*);

	4	*if*(*MARK*(*memory_chips*) == *39*)
		  return(*0.0025*);
		else
		  return(*0.000005*);

definitions may be entered exactly as they are shown in the tables. The tables of component definitions in this paper have been created by the automatic documentation facility that is available from the control panel of *UltraSAN*. The tables are written in LaTeX and are generated from the model description files.

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

Composed Model Specification

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

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

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

The composed model for the multicomputer system is shown in Figure B.4. As can be seen from the figure, a model of a single computer is composed by joining three replicas of the memory module model with models of the CPU modules, I/O

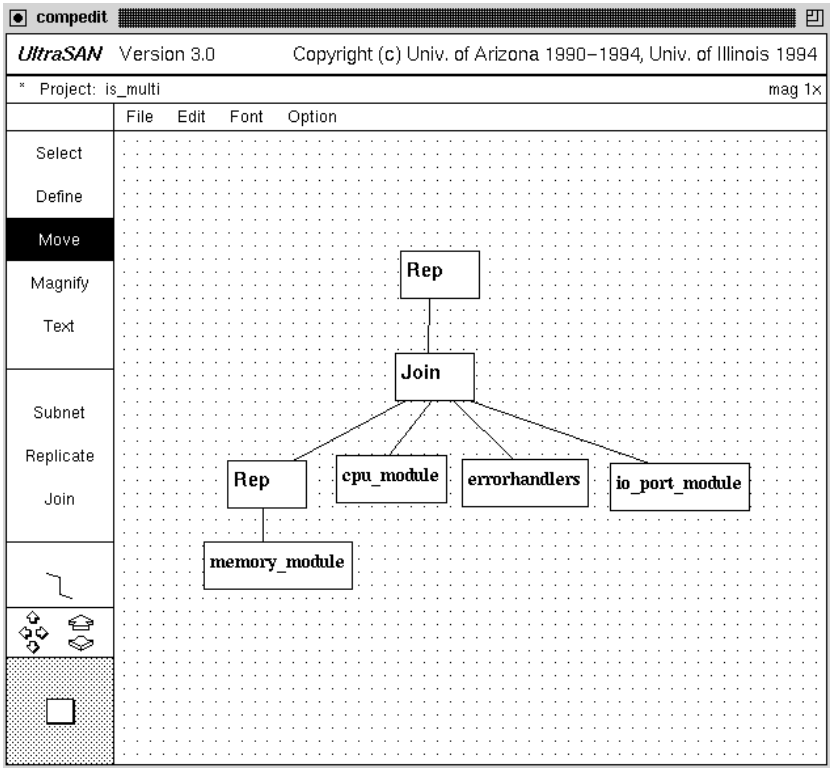


Figure B.4: Composed model for multicomputer, as specified in the composed model editor.

ports, and the errorhandler. The computer model is then replicated to produce a model of the multicomputer system. The place *computer_failed* is an example of a common place. It is held common at the system level, where it can be accessed by all subsystems, so that they may update the number of failed computers in the event of a coverage failure.

Performance, Dependability, and Performability Variable Specification

The third step in model specification is to use the performability variable editor to build reward structures from which the desired performance measures may be derived. In short, to specify a performability variable one defines a reward structure and chooses a collection policy. Variables are typed and often referred to according to the collection policy. The three types of variables supported in *UltraSAN* are "instant-of-time," "interval-of-time," and "time-averaged interval-of-time." *Instant-of-time* variables are obtained by examining the value of the associated reward structure at a particular time t. *Interval-of-time* variables measure the reward accumulated in an interval $[t, t + l]$. *Time-averaged interval-of-time* variables divide the accumulated reward by the length of the interval. For a thorough discussion of the concept and application of these variables, see [79].

For simulation models, "activity variables" can also be defined. *Activity variables* measure the time between completions of activities. Using an activity variable, one can measure the mean and variance of the time until first completion, the time between the tenth and twentieth completions, etc., using the terminating simulator. The steady-state simulator can be used to measure the mean and variance of the time between consecutive completions as time goes to infinity.

Importance Sampling Governor Specification

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

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

In this way, importance sampling can be used to focus the simulation effort on sample paths leading to the event. The stochastic components of the simulation model are altered to increase the likelihood of encountering the rare event. A good example of a problem where importance sampling can help is estimating the unreliability of dependable systems [46, 65, 67]. In this case, the model is altered to accelerate component failures with the intent of increasing the probability of system failure. The cited research shows the utility of varying the acceleration applied to component failures depending on the evolution of a sample path. This approach is one heuristic for evaluating dependable systems. For simulations of other types of systems, different heuristics will be required, since no universally applicable heuristic has been discovered.

In *UltraSAN*, an importance sampling governor [68] is used to specify the heuristic. The governor is similar to a finite state machine, where each state corresponds to a different set of activity definitions. That is to say, in each governor state the user may redefine the activity time and case distribution of each timed activity in each SAN in the composed model. There are two restrictions that must be met for the simulation to be valid. All activity time distribution functions must have closed form complementary distribution functions, and case distributions cannot be altered to give zero probability to cases that have positive probability in the original model. Both of these conditions are checked by the importance sampling simulator at run time.

The governor changes state based on the evolution of the simulation. Each state has a transition function consisting of a sequence of Boolean functions of the markings of the SANs in the composed model (called predicates), paired with a governor state name. When the marking of the model is updated, the transition function for the current governor state is checked. The first predicate that holds causes the governor to change to the associated governor state. When the governor changes state, the activity definitions associated with the new state are applied to the model. Activities whose definitions are changed by the governor transition are reactivated so that the new definition takes effect immediately. When an activity is reactivated its completion time is sampled from the new activity time distribution function. The details of this process are available in [68].

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

This approach differs from that taken in other tools, which are specialized to apply a single heuristic. The automatic application of a single heuristic is a feature

found in more specialized tools such as SAVE [6], which is designed to handle Markovian dependability models. SAVE applies balanced failure biasing [88], a provably robust technique for importance sampling simulation of models representable in the SAVE model description language. Since *UltraSAN* is intended to handle a broader class of systems, the use of a single heuristic for automatic importance sampling is not practical.

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

B.5 Global Variable Assignment

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

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

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

specified for all global variables. Despite its limited scope, the range editor is very efficient for generating experiments for many of studies. When more flexibility is required, the set editor may be used.

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

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

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

B.6 Study Construction

Depending on model characteristics, experiments in a study can be solved by analysis or simulation. Specifically, analytic solution can be used when activity time distributions are exponential, activities are reactivated often enough to ensure that their rates depend only on the current state, and the state space is not too

large relative to the capacity of the machine. In this case, the underlying stochastic process for the model is a Markov process. In addition, mixes of exponential and deterministic activities can be handled, as long as no more than one concurrently enabled deterministic activity is enabled in any reachable stable marking. Otherwise, simulation must be used.

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

One potential solution to this problem is to construct a process which keeps track (in its notion of state) of most recently completed activities, as well as reached stable markings. This process (called the activity-marking behavior, in [85]), supports all reward variables, but will result in state spaces that are very large. Alternatively, one can construct a stochastic process that is tailored to the variable in question, and exploits symmetries inherent in the model. When such symmetries occur, this approach, known as "reduced base model construction" can significantly reduce the size of the process that must be solved to obtain a solution. Symmetries are identified through the use of the replicate and join operation in the composed model editor (as described in the model specification section). This resulting state-level representation is known as a *reduced base model* [85].

The reduced base model is constructed from the model specification, which is known formally as a "composed SAN-based reward model." In order to understand

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

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

Composition of multiple distinct SBRMs is done using the join operation. The *join* operation acts on SBRMs and produces a SBRM. The new SBRM is produced by identifying certain places in the different input SANs as common in the new

SAN. These common places allow arbitrary communication between the SANs that are joined; the only restriction being that the initial marking of the places to be combined is identical. The reward structure in the new SBRM is constructed by assigning: 1) an impulse reward to each activity in the joined SBRM equal to the reward of the corresponding activity in the input SBRM, and 2) a rate reward to each marking in the joined SBRM equal to the sum of the rate rewards of the markings in the input SBRM whose union constitutes the marking of the joined SBRM. The set of common places of new SBRM is the subset of the set of common places of each input SAN-based reward model that is made common, with zero or more models that are joined together.

The composed SAN-based reward model constructed in this way is input to the reduced base model construction procedure to generate the reduced base model. The notion of state employed is variable and depends on the structure of the composed model. One can think of the state as a set of impulse and rate rewards plus a state tree [81], where each node on the state tree corresponds in type and level with a node on the composed model tree (as specified in the composed model editor). Each node in a state tree has associated with it a subset of common places of the corresponding node in the composed model diagram. These places are those that are common at the node, but not at its parent node. The saving in state space size, relative to the standard state generation approach, is due to the fact that at each replicate node in the tree, we keep track of the *number* of submodels in particular markings, rather than the marking of *each* submodel. Proof that this notion of state produces Markov processes whenever the standard state generation algorithm can be found in [85].

Figure B.5 shows the state tree of a particular state of the reduced base model for the multicomputer example. Places *interface_chips* and *memory_chips* are not common at the replicate node *R2*. Therefore, as shown in Figure B.5, they are

unique for each memory module, and can be different for different replica copies. At the join node, the set of common places is the union of the common places associated with the SANs of cpu_module, errorhandlers, io_port_module, and replicate node R2. Finally, among these places, *computer_failed* is common to all computer modules. The other places: *memory_failed*, *cpus*, *errorhandlers*, and *ioports* are not common, and hence unique to each computer module.

A reduced base model employing this notion of state can be generated directly without first generating the detailed (marking or activity-marking) behavior. This is done by first creating the state tree corresponding to the initial marking of the SAN, and then generating all possible next states by executing each activity that may complete in the state. A new state tree and impulse reward are generated corresponding to each possible next state that may be reached. For each possible next state, a non-zero rate (or probability, if a deterministic activity is enabled in the state, see [87]) from the original state to the reached state is added to the set of rates to other states from the originating state. If the reached state is new, then it is added to the set of states which need to be explored. Generation of the reduced base model proceeds by selecting states from the set of unexplored states and repeating the above operations. The procedure terminates when there are no more states to explore. The resulting reduced base model serves as input to the analytic solvers, which determine the probabilistic nature of the selected performability variables.

Unlike the analytic solvers, which require a reduced base model, the simulators execute directly on the assigned model specifications. As depicted in Figure B.1, three types of simulation programs can be generated: a *steady-state simulator* to solve long-run measures of performability variables, a *terminating simulator* to solve instant-of-time or interval-of-time performability variables, and an importance sampling terminating simulator. When simulation is the intended solution method, in

the study construction phase the control panel links experiments (model descriptions) with the appropriate simulation library to generate the selected executable simulation programs. In the case of importance sampling simulation, the control panel links the governor description created by the importance sampling editor together with the experiment and importance sampling simulation library.

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

B.7 Study Solution

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

B.7.1 Analytic Solution Techniques

Analytic solvers can provide steady-state as well as transient solutions for the reduced base model generated during study construction. Figure B.1 illustrates the six solvers available for analytic solutions. Three of the solvers, specifically, the *direct steady-state solver*, *iterative steady-state solver*, and *deterministic iterative steady-state solver* provide steady-state solutions for the instant-of-time variables. The other three solvers, the *transient instant-of-time solver*, *PDF interval-of-time solver* and *expected interval-of-time solver*, provide solutions for transient instant-of-time and interval-of-time variables. Except for the two interval-of-time solvers,

each solver calculates the mean, variance, probability density function, and probability distribution function of the variable(s) for which it is intended to solve. The PDF interval-of-time solver calculates the probability distribution function of reward accumulated during a fixed interval of time $[0, t]$. Likewise, the expected interval-of-time solver calculates the expected reward accumulated during a fixed interval of time. All solvers use a sparse matrix representation appropriate for the solution method employed.

The direct steady-state solver uses LU decomposition [95] to obtain the steady-state solution for instant-of-time variables. As with all steady-state solvers, it is applicable when the generated reduced base model is Markovian, and contains a single, irreducible, class of states. LU decomposition factors the transition matrix Q into the product of a lower triangular matrix L and an upper triangular matrix U such that $Q = LU$. Crout's algorithm [75] is incorporated to compute L and U in a memory efficient way. Pivoting in the solver is performed by using the improved generalized Markowitz strategy, as discussed by Osterby and Zlatev [69]. This strategy performs pivoting on the element for which the minimum fill-in is expected, subject to constraints on the magnitude of the pivot. Furthermore, the matrix elements that are less than some value (called the drop tolerance) are made zero (i.e., *dropped*) to retain the sparsity of the matrix during decomposition. In the end, iterative refinement is used to correct the final solution to a user-specified level of accuracy.

The iterative steady-state solver uses successive-overrelaxation to obtain steady-state instant-of-time solutions. Like the direct steady-state solver, it acts on the Markov process representation of a reduced base model. However, being an iterative method, it avoids fill-in problems inherent to the direct steady-state solver and therefore requires less memory. However, unlike LU decomposition, SOR is not guaranteed to converge to the correct solution for all models. In practice however,

this does not seem to be a problem. The implementation solves the system of equations directly, without first adding the constraint that the probabilities sum to one, and normalizes only at the end and as necessary to keep the solution vector within bounds, as suggested in [42]. Selection of the acceleration factor is left to the user and defaults to a value of 1 (resulting in a Gauss-Seidel iteration).

The method used in the deterministic iterative steady-state solver was first developed for deterministic stochastic Petri nets [47, 50] with one concurrently-enabled deterministic transition, and later adapted to SANs in which there is no more than one concurrently-enabled deterministic activity [87]. In this solution method, a Markov chain is generated for marking in which a deterministic activity is enabled, representing the states reachable from the current state due to completions of exponential activities, until the deterministic activity either completes or is aborted. The solver makes use of successive-overrelaxation and uniformization to obtain the solution.

The transient instant-of-time solver and expected interval-of-time solver both use uniformization to obtain solutions. Uniformization works well when the time point of interest is not too large, relative to the highest rate activity in the model. The required Poisson probabilities are calculated using the method by Fox and Glynn [31] to avoid numerical difficulties.

The PDF interval-of-time solver [76] also uses uniformization to calculate the probability distribution function of the total reward accumulated during a fixed interval $[0, t]$. This solver is unique among the existing interval-of-time solvers in that it can solve for reward variables containing both impulse and rate rewards. The solver calculates the PDF of reward accumulated during an interval by conditioning on possible numbers of transitions that may occur in an interval, and possible sequences of state transitions (paths), given a certain number of transitions have occurred. Since the number of possible paths may be very large, an

implementation that considers all possible paths would require a very large amount of memory. This problem is avoided in *UltraSAN* by calculating a bound on the error induced in the desired measure by not considering each path, and discarding those with acceptably small (user specified) errors. In many cases as illustrated in [76], selective discarding of paths can dramatically reduce the space required for a solution while maintaining acceptable accuracy. Calculation of the PDF of reward accumulated, given that a particular path was taken, can also pose numerical difficulties. In this case, the solver makes selective use of multiprecision math to calculate several summations. Experience with the implementation has shown that this selective use of multiprecision operations avoids numerical difficulties, while maintaining acceptably fast execution.

B.7.2 Simulative Solutions

As shown in Figure B.1, three simulation-based solvers are available in *UltraSAN.* All three simulators exploit the hierarchical structure and symmetries introduced by the replicate operation in a composed SAN-based reward model to reduce the cost of future event list management. More specifically, multiple future events lists are employed, one corresponding to each leaf on the state tree or, in other words one corresponding to each set of replica submodels in a particular marking. These multiple future event lists allow the simulators to operate on "compound events," corresponding to a set of enabled replica activities, rather than on individual activities. By operating on compound events, rather than individual activities, the number of checks that must be made upon each activity completion is reduced. The details and a precise algorithm can be found in [81]. This algorithm is the basis of the state-change mechanism for all three simulators.

The steady-state simulator uses an iterative batch means method to estimate the means and/or variances of steady-state instant-of-time variables. The user selects

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

The third simulation-based solver is a terminating simulator that is designed to use importance sampling to increase the efficiency of simulation-based evaluation of rare event probabilities. The importance sampling terminating simulator estimates the mean of transient variables that are affected by rare events by simulating the model under the influence of the governor (described in Section B.4), so that the interesting event is no longer rare. The simulator then compensates for the governor bias by weighting the observations by the likelihood ratio. The likelihood ratio is calculated by evaluating the ratio of the likelihood that a particular activity completes in a marking, and that the completion of this activity results in a particular next marking, in the original model to the likelihood of the same event in the governed model. The likelihood calculation is based on the work in [66], adapted for SAN simulation and implemented in *UltraSAN* [68]. It works for models where all activities have closed-form activity time distribution functions. The probability distribution function is needed to handle the reactivations of activities when their definitions are changed by the governor.

B.8 Example Results

The reliability of the multicomputer system example discussed earlier has been evaluated in [44] using a hierarchical evaluation technique based on numerical integration, and using SANs and the *UltraSAN* transient instant-of-time solver in [83].

Both studies focus on long mission times. In this section we discuss the evaluation of the unreliability of the multicomputer for shorter mission times, using the transient instant-of-time solver and the importance sampling terminating simulator.

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

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

If importance sampling is to be used to evaluate the unreliability, a governor must be constructed to guide the importance sampling simulation. In this case, we chose a heuristic called "approximate forcing with balanced failure biasing [65]." Approximate forcing is based on the forcing technique developed in [46] to ensure that a failure event occurs before the mission is completed. In approximate forcing, failure events are accelerated so that a component failure occurs before the end of a mission with probability 0.8. When combined with balanced failure biasing, not only is the total probability of a failure event before mission completion increased, but the conditional probabilities of particular failure events, given a failure event occurs, are made equal.

A governor that implements approximate forcing with balanced failure biasing consists of two states. In the first state, all failure events are given equal probability

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

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

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

The multicomputer model with two computer modules was evaluated for missions from one to 390 days in length, using the transient instant-of-time solver and the importance sampling terminating simulator with the governor described above. The accuracy requested in the solver was 1×10^{-9}, while the simulator was set to run until the estimated half width of a 99% confidence interval was within 10% of the point estimate. The results from the two different solution approaches are plotted

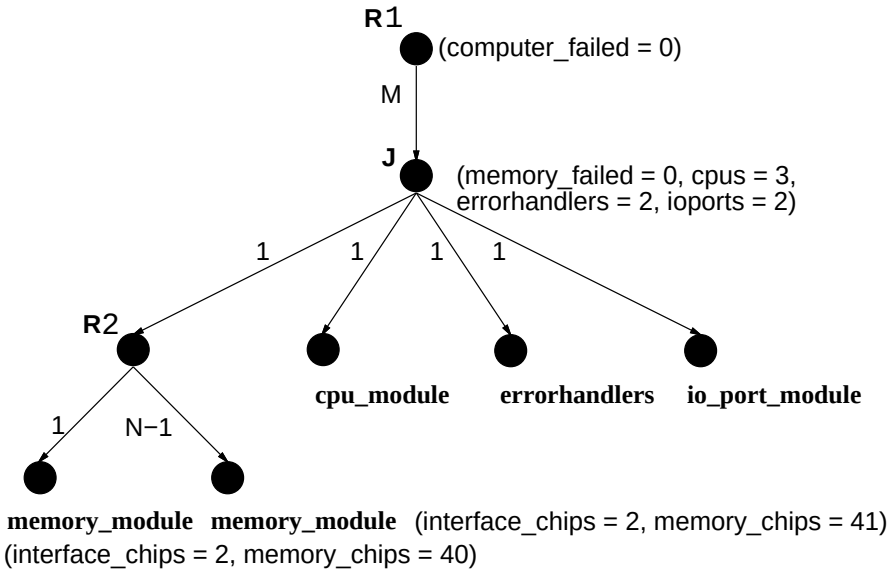


Figure B.5: The state tree representation

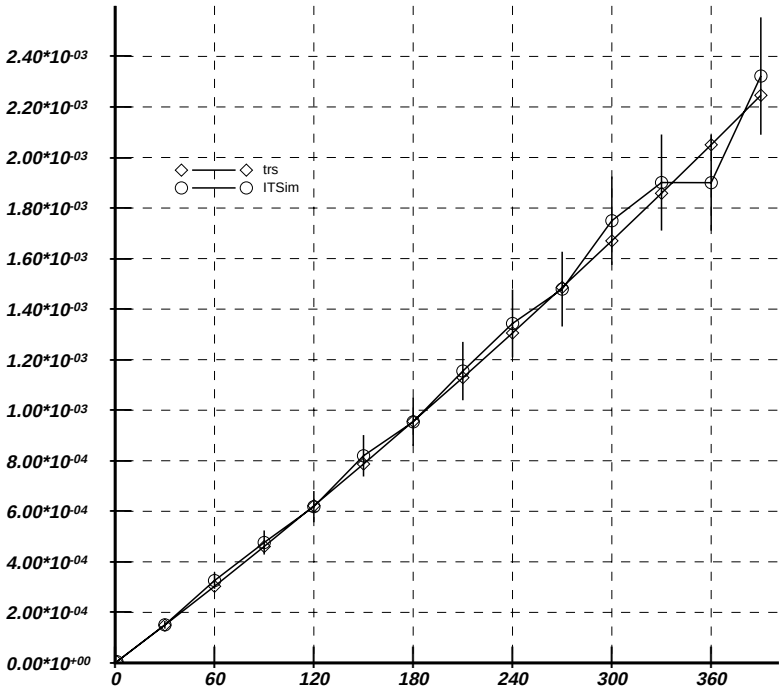


Figure B.6: Unreliability versus mission time (in days).

in Figure B.6. As can be seen from the figure, the results are in close agreement. There is one errant point from the simulator, but the result obtained from the transient instant-of-time solver is still within the confidence interval.

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

The impact of perturbations in the coverage factors for RAM chip failures and computer module failures was evaluated using the transient instant-of-time solver for the multicomputer with two computer modules. These studies were carried out using the study definitions shown in Table B.5. As can be seen from Figure B.8, the unreliability is not very sensitive to variations in the RAM chip failure coverage. However, variations in the coverage of computer module failures had a large impact on the unreliability, as shown in Figure B.9.

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

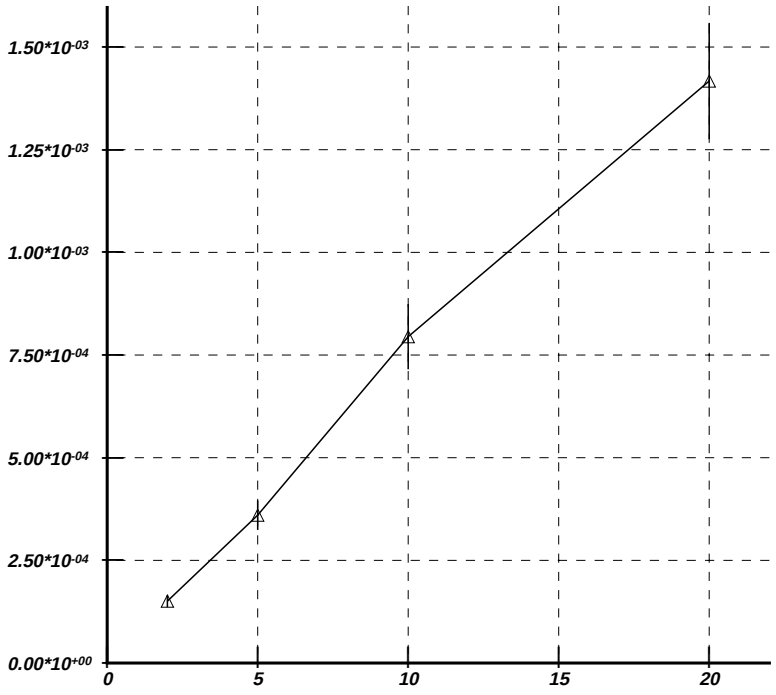


Figure B.7: Unreliability versus number of computer modules.

Table B.5: Study Editor Range Definitions for Project *multi_proc*

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

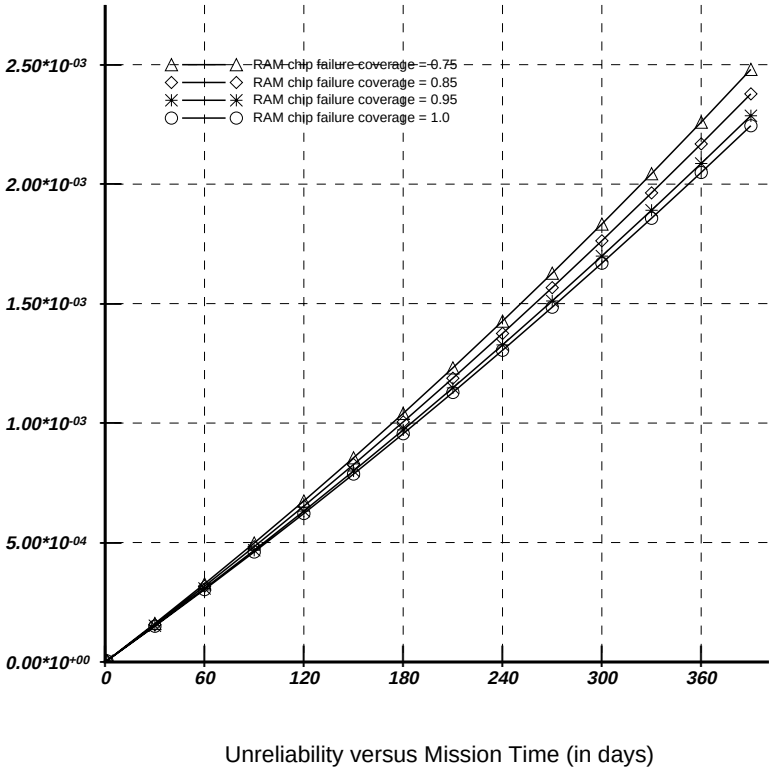


Figure B.8: Impact of perturbation in RAM chip failure coverage factor.

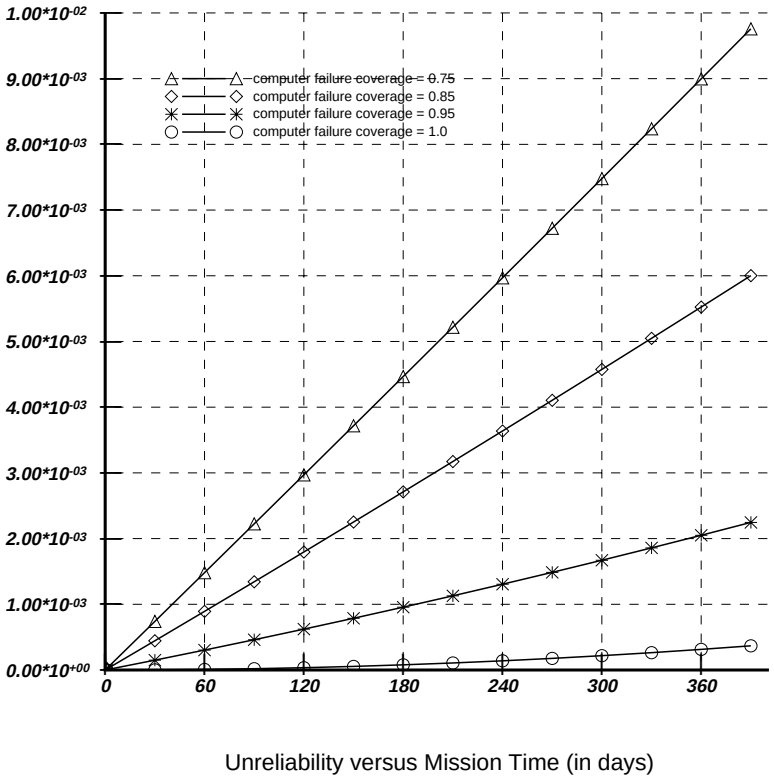


Figure B.9: Impact of perturbation in computer failure coverage factor.

B.9 Conclusions

We have described *UltraSAN*, an environment designed to facilitate the specification, construction, and solution of performability models represented as composed stochastic activity networks. *UltraSAN* has been in use for a period of about four years at several industrial and academic sites. Feedback from these users, and the development of new solution methods, has motivated continuous improvement of the software. As it stands today, *UltraSAN* is an environment that incorporates features and capabilities that do much to automate and simplify the task of model-based evaluation, and as evidenced by its application (e.g., [49, 55, 82, 83]), is capable of evaluating the performance, dependability, and performability of "real-world" systems. In addition to this capability, the tool serves as a test-bed in the development of new model construction and solution algorithms.

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

In the area of model construction, reduced base model construction allows *UltraSAN* to handle the typically large state spaces of models of "real-world" systems.

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

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

Acknowledgments

The success of this project is the result of the work of many people, in addition to the authors of this paper. In particular we would like to thank Burak Bulasaygun, Bruce McLeod, Aad van Moorsel, Bhavan Shah, and Adrian Suherman, for their contributions to the tool itself, and the members of the Performability Modeling Research Lab who tested the pre-release versions of the software. We would also like to thank the users of the released versions who made suggestions that helped

improve the software. In particular, we would like to thank Steve West of IBM Corporation, Dave Krueger, Renee Langefels, and Tom Mihm of Motorola Satellite Communications, Kin Chan, John Chang, Ron Leighton and Kishore Patnam of US West Advanced Technologies, Lorenz Lercher of the Technical University of Vienna, and John Meyer, of the University of Michigan in this regard. We would also like to thank Sue Windsor for her careful reading of the manuscript.

REFERENCES

[1] B. E. Aupperle and J. F. Meyer, "Fault-tolerant BIBD networks," in *18th International Symposium on Fault-Tolerant Computing (FTCS-18)*, pp. 306–311, Tokyo, Japan, June 27–30 1988.

[2] B. E. Aupperle and J. F. Meyer, "State space generation for degradable multiprocessor systems," in *21st International Symposium on Fault-Tolerant Computing (FTCS-21)*, pp. 308–315, Montréal, Canada, June 25–27 1991.

[3] L. Babai, E. M. Luks, and Ákos Seress, "Fast management of permutation groups I," *SIAM Journal on Computing*, vol. 26, no. 3, pp. 1310–1342, October 1997.

[4] M. D. Beaudry, "Performance related reliability measures for computing systems," in *Proceedings of the Seventh Annual International Symposium on Fault-Tolerant Computing*, pp. 16–21, 1977.

[5] C. Béounes, M. Aguéra, J. Arlat, S. Bachmann, C. Bourdeau, J.-E. Doucet, K. Kanoun, J.-C. Laprie, S. Metge, J. M. de Souza, D. Powell, and P. Spiesser, "SURF-2: A program for dependability evaluation of complex hardware and software systems," in *Proceedings of the Twenty-Third International Symposium on Fault-Tolerant Computing*, pp. 668–673, Toulouse, France, June 1993.

[6] A. M. Blum, P. Heidelberger, S. S. Lavenberg, M. Nakayama, and P. Shahabuddin, "System availability estimator (SAVE) language reference and user's manual version 4.0," Technical Report RA 219 S, IBM Thomas J. Watson Research Center, June 1993.

[7] M. Bouissou, H. Bouhadana, M. Bannelier, and N. Villatte, "Knowledge modeling and reliability processing: The FIGARO language and associated tools," in *Proceedings of the Twenty-Third Annual International Symposium on Fault-Tolerant Computing*, pp. 680–685, Toulouse, France, June 1993.

[8] P. Buchholz, "Hierarchical Markovian models: Symmetries and reduction," *Performance Evaluation*, vol. 22, no. 1, pp. 93–110, February 1995.

[9] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper, "Complexity of Kronecker operations on sparse matrices with applications to solution of Markov models," ICASE Report #97-66 (NASA/CR-97-206274), NASA Langley Research Center, December 1997.

[10] P. Buchholz and P. Kemper, "Numerical analysis of stochastic marked graph nets," in *Proceedings of the Sixth International Workshop on Petri Nets and Performance Models*, pp. 32–41, Los Alamitos, CA, 1995, IEEE Computer Society Press. Cat. No.95TB100003.

[11] G. Butler, *Fundamental Algorithms for Permutation Groups*, volume 559 of *Lecture Notes in Computer Science*, Springer-Verlag, 1991.

[12] J. A. Carrasco, "Automated construction of compound Markov chains from generalized stochastic high-level Petri nets," in *Proceedings of the Third International Workshop on Petri Nets and Performance Models*, pp. 93–102, Kyoto, Japan, December 11–13 1989.

[13] J. A. Carrasco, J. Escriba, and A. Calderon, "Efficient exploration of availability models guided by failure distances," *Performance Evaluation Review*, vol. 24, no. 1, pp. 242–251, May 1996.

[14] *UltraSAN Reference Manual*, Center for Reliable and High Performance Computing, Coordinated Science Laboratory, University of Illinois, 1995.

[15] G. Chiola, "A software package for analysis of generalized stochastic Petri net models," in *Proceedings of the International Workshop on Timed Petri Net Models*, pp. 136–143, Torino, Italy, July 1985.

[16] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad, "Stochastic well-formed colored nets and symmetric modeling applications," *IEEE Transactions on Computers*, vol. 42, no. 11, pp. 1343–1360, November 1993.

[17] G. Ciardo, A. Blakemore, P. F. J. Chimento, J. K. Muppala, and K. S. Trivedi, "Automated generation and analysis of Markov reward models using stochastic reward nets," in *IMA Volumes in Mathematics and its Applications*, Springer-Verlag, New York, 1992. vol. 48.

[18] G. Ciardo, J. Muppala, and K. S. Trivedi, "SPNP: Stochastic Petri net package," in *Proceedings of the Fourth International Workshop on Petri Nets and Performance Models*, pp. 142–151, Kyoto, Japan, December 1989.

[19] G. Ciardo and M. Tilgner, "On the use of Kronecker operators for the solution of generalized stochastic petri nets," ICASE Report #96-35 (NASA CR-198336), NASA Langley Research Center, May 1996.

[20] G. Ciardo and M. Tilgner, "Parametric state space structuring," ICASE Report #97-67 (NASA/CR-97-206267), NASA Langley Research Center, December 1997.

[21] G. Ciardo and K. S. Trivedi, "A decomposition approach for stochastic reward net models," *Performance Evaluation*, vol. 18, pp. 37–59, 1993.

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

[23] J. Couvillion, R. Freire, R. Johnson, W. D. Obal II, M. A. Qureshi, M. Rai, W. H. Sanders, and J. Tvedt, "Performability modeling with *UltraSAN*," *IEEE Software*, vol. 8, no. 5, pp. 69–80, September 1991.

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

[25] E. de Souza e Silva and P. M. Ochoa, "State space exploration in Markov models," *Performance Evaluation Review*, vol. 20, no. 1, pp. 152–166, June 1992.

[26] D. Deavours and W. H. Sanders, "An efficient disk-based tool for solving very large Markov models," in *Computer Performance Evaluation: Proceedings of the 9th International Conference on Modelling Techniques and Tools (TOOLS '97)*, pp. 58–71, St. Malo, France, June 3–6 1997. Lecture Notes in Computer Science, no. 1245.

[27] D. Deavours and W. H. Sanders, ""On-the-Fly" solution techniques for stochastic Petri nets and extensions," in *Proceedings of the 7th International Workshop on Petri Nets and Performance Models (PNPM '97)*, pp. 132–141, St. Malo, France, June 3–6 1997.

[28] S. Donatelli, "Superposed stochastic automata: A class of stochastic Petri nets with parallel solution and distributed state space," *Performance Evaluation*, vol. 18, no. 1, pp. 21–36, July 1993.

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

[30] G. Florin, P. Lonc, S. Natkin, and J. M. Toudic, "RDPS: A software package for the validation and evaluation of dependable computer systems," in *Proceedings of the Fifth IFAC Workshop Safety of Computer Control Systems (SAFECOMP'86)*, W. J. Quirk, editor, pp. 165–170, Pergamon, Sarlat, France, October 1986.

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

[32] G. Franceschinis and R. R. Muntz, "Bounds for quasi-lumpable Markov chains," *Performance Evaluation*, vol. 20, no. 1–3, pp. 223–243, May 1994.

[33] G. Franceschinis and R. Muntz, "Computing bounds for the performance indices of quasi-lumpable stochastic well-formed nets," *IEEE Transactions on Software Engineering*, vol. 20, no. 7, pp. 516–525, July 1994.

[34] R. Geist and K. Trivedi, "Reliability estimation of fault-tolerant systems: Tools and techniques," *Computer*, pp. 52–61, July 1990.

[35] R. German, C. Kelling, A. Zimmermann, and G. Hommel, "TimeNET: A toolkit for evaluating non-Markovian stochastic petri-nets," Technical report, Technische Universitat Berlin, Germany, 1994. Submitted for publication.

[36] B. R. Haverkort and I. G. Niemegeers, "Performability modelling tools and techniques," *Performance Evaluation*, 1994.

[37] B. R. Haverkort, "In search of probability mass: Probabilistic evaluation of high-level specified Markov models," *The Computer Journal*, vol. 38, no. 7, pp. 521–529, 1995.

[38] B. R. H. M. Haverkort, *Performability Modelling Tools, Evaluation Techniques, and Applications*, PhD thesis, Universiteit Twente, 1990.

[39] J. Hillston, *A Compositional Approach to Performance Modelling*, Distinguished Dissertations in Computer Science, Cambridge University Press, Cambridge, Great Britain, 1996.

[40] R. A. Howard, *Dynamic Probabilistic Systems, Vol. II: Semi-Markov and Decision Processes*, Wiley, New York, 1971.

[41] P. Kemper, "Numerical analysis of superposed GSPNs," *IEEE Transactions on Software Engineering*, vol. 22, no. 9, pp. 615–628, September 1996.

[42] U. R. Krieger, B. Müller-Clostermann, and M. Sczittnick, "Modeling and analysis of communication systems based on computational m ethods for Markov chains," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 9, pp. 1630–1648, 1990.

[43] J.-C. Laprie, "Dependable computing and fault-tolerance: Concepts and terminology," in *Proceedings of the Fifteenth Annual International Symposium on Fault-Tolerant Computing (FTCS-15)*, pp. 2–11, 1985.

[44] D. Lee, J. Abraham, D. Rennels, and G. Gilley, "A numerical technique for the hierarchical evaluation of large, closed fault-tolerant systems," in *Dependable Computing for Critical Applications 2*, J. F. Meyer and R. D. Schlicting, editors, volume 6 of *Dependable Computing and Fault-Tolerant Systems (eds. A. Avižienis, H. Kopetz, J. C. Laprie)*, Springer-Verlag, New York, 1992.

[45] J. S. Leon, "Permutation group algorithms based on partitions, I: Theory and algorithms," *Journal of Symbolic Computation*, vol. 12, pp. 533–583, 1991.

[46] E. E. Lewis and F. Böhm, "Monte Carlo simulation of Markov unreliability models," *Nuclear Engineering and Design*, vol. 77, pp. 49–62, January 1984.

[47] C. Lindemann, "An improved numerical algorithm for calculating steady-state solutions of deterministic and stochastic Petri net models," in *Proceedings of the Fourth International Workshop on Petri Nets and Performance Models*, pp. 176–185, Melbourne, Australia, 1991.

[48] C. Lindemann, "DSPNexpress: A software package for the efficient solution of deterministic and stochastic Petri nets," in *Proceedings of the Sixth International Conference on Modelling Techniques and Tools for Computer Systems Performance Evaluation*, pp. 15–29, Edinburgh, Great Britain, 1992.

[49] L. M. Malhis, W. H. Sanders, and R. D. Schlichting, "Analytic performability evaluation of a group-oriented multicast protocol," Technical Report PMRL 93-13, The University of Arizona, June 1993. Submitted for publication.

[50] M. A. Marsan and G. Chiola, "On Petri nets with deterministic and exponentially distributed firing times," in *Advances in Petri Nets 1986*, G. Rozenberg, editor, Lecture Notes in Computer Science 266, pp. 132–145, Springer, 1987.

[51] J. Marshall Hall, *The Theory of Groups*, Chelsea Publishing Company, New York, second edition, 1976.

[52] L. C. Mathewson, *Elementary Theory of Finite Groups*, Houghton Mifflin Company, Boston, 1930.

[53] B. D. McKay, "Practical graph isomorphism," *Congressus Numerantium*, vol. 30, pp. 45–87, 1981.

[54] B. D. McKay, "Nauty users guide (version 1.5)," Technical Report TR-CS-90-02, Computer Science Department, Australian National University, 1990.

[55] B. D. McLeod and W. H. Sanders, "Performance evaluation of N-processor time warp using stochastic activity networks," Technical Report PMRL 93-7, The University of Arizona, March 1993. Submitted for publication.

[56] J. F. Meyer, "On evaluating the performability of degradable computing systems," in *Proceedings of the Eighth Annual International Symposium on Fault-Tolerant Computing (FTCS-8)*, pp. 44–49, 1978.

[57] J. F. Meyer, "On evaluating the performability of degradable computing systems," *IEEE Transactions on Computers*, vol. C-22, pp. 720–731, August 1980.

[58] J. F. Meyer, "Probabilistic modeling," in *The Twenty-Fifth International Symposium on Fault-Tolerant Computing (FTCS-25)*, pp. 88–95, Pasadena, California, June 27–30, 1995.

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

[60] M. K. Molloy, "Performance analysis using stochastic Petri nets," *IEEE Transactions on Computers*, vol. C-31, pp. 913–917, September 1982.

[61] A. Movaghar and J. F. Meyer, "Performability modeling with stochastic activity networks," in *Proceedings of 1984 Real-Time Systems Symposium*, pp. 215–224, Austin, Texas, December 1984.

[62] H. Nabli and B. Sericola, "Performability analysis: A new algorithm," *IEEE Transactions on Computers*, vol. 45, no. 4, pp. 491–494, April 1996.

[63] S. Natkin, *Reseaux de Petri Stochastiques*, PhD thesis, CNAM-PARIS, June 1980.

[64] M. F. Neuts, *Matrix-Geometric Solutions in Stochastic Models: An Algorithmic Approach*, Johns Hopkins, Baltimore, Maryland, 1981.

[65] V. F. Nicola, P. Heidelberger, and P. Shahabuddin, "Uniformization and exponential transformation: Techniques for fast simulation of highly dependable non-Markovian systems," in *Proceedings of the Twenty-Second Annual International Symposium on Fault-Tolerant Computing*, pp. 130–139, Boston, Massachusetts, 1992.

[66] V. F. Nicola, M. K. Nakayama, P. Heidelberger, and A. Goyal, "Fast simulation of dependability models with general failure, repair and maintenance processes," in *Proceedings of the Twentieth Annual International Symposium on Fault-Tolerant Computing*, pp. 491–498, Newcastle upon Tyne, United Kingdom, June 1990.

[67] W. D. Obal II and W. H. Sanders, "An environment for importance sampling based on stochastic activity net works," in *Proceedings of the 13th Symposium on Reliable Distributed Systems*, pp. 64–73, Dana Point, California, October 1994.

[68] W. D. Obal II and W. H. Sanders, "Importance sampling simulation in *UltraSAN*," *Simulation*, vol. 62, no. 2, pp. 98–111, February 1994.

[69] O. Osterby and Z. Zlatev, *Lecture Notes in Computer Science: Direct Methods for Sparse Matrices*, Springer-Verlag, Heidelberg, 1983.

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

[71] K. T. Pedretti and S. A. Fineberg, "Analysis of 2D torus and hub topologies of 100Mb/s ethernet for the Whitney commodity computing testbed," Technical Report NAS-97-017, Numerical Aerodynamic Simulation, NASA Ames Research Center, September 1997.

[72] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, N.J., 1981.

[73] B. Plateau and K. Atif, "Stochastic automata network for modeling parallel systems," *IEEE Transactions on Software Engineering*, vol. 17, no. 10, pp. 1093–1108, October 1991.

[74] B. Plateau and J.-M. Fourneau, "A methodology for solving Markov models of parallel systems," *Journal of Parallel and Distributed Computing*, vol. 12, no. 4, pp. 370–387, August 1991.

[75] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, 1988.

[76] M. A. Qureshi and W. H. Sanders, "Reward model solution methods with impulse and rate rewards: An algorit hm and numerical results," *Performance Evaluation*, vol. 20, pp. 413–436, August 1994.

[77] M. A. Qureshi and W. H. Sanders, "A new methodology for calculating distributions of reward accumulated during a finite interval," in *Proceedings of the 26th International Symposium on Fault-Tolerant Computing (FTCS-26)*, pp. 116–125, Sendai, Japan, June 25–27, 1996.

[78] M. A. Qureshi, W. H. Sanders, A. P. A. van Moorsel, and R. German, "Algorithms for the generation of state-level representations of stochastic activity networks with general reward structures," *IEEE Transactions on Software Engineering*, vol. 22, no. 9, pp. 603–614, September 1996.

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

[80] W. H. Sanders, *Construction and Solution of Performability Models Based on Stochastic Activity Networks*, PhD thesis, University of Michigan, 1988.

[81] W. H. Sanders and R. S. Freire, "Efficient simulation of hierarchical stochastic activity network models," *Discrete Event Dynamic Systems: Theory and Applications*, vol. 3, no. 2/3, pp. 271–300, July 1993.

[82] W. H. Sanders, L. A. Kant, and A. Kudrimoti, "A modular method for evaluating the performance of picture archiving and communication systems," *Journal of Digital Imaging*, vol. 6, no. 3, pp. 172–193, August 1993.

[83] W. H. Sanders and L. M. Malhis, "Dependability evaluation using composed SAN-based reward models," *Journal of Parallel and Distributed Computing*, vol. 15, no. 3, pp. 238–254, July 1992.

[84] W. H. Sanders and J. F. Meyer, "METASAN: A performability evaluation tool based on stochastic activity networks," in *Proceedings of the IEEE-ACM Fall Joint Computer Conference*, pp. 807–816, Dallas, Texas, November 1986.

[85] W. H. Sanders and J. F. Meyer, "Reduced base model construction methods for stochastic activity networks," *IEEE Journal on Selected Areas in Communication*, vol. 9, no. 1, pp. 25–36, January 1991.

[86] M. Schönert et al., *GAP – Groups, Algorithms, and Programming*, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, fifth edition, 1995.

[87] B. P. Shah, *Analytic Solution of Stochastic Activity Networks with Exponential and Deterministic Activities*, Master's thesis, The University of Arizona, 1993.

[88] P. Shahabuddin, *Simulation and Analysis of Highly Reliable Systems*, PhD thesis, Stanford University, 1990.

[89] M. Siegle, "Reduced Markov models of parallel programs with replicated processes," in *Proceedings of the Second Euromicro Workshop on Parallel and Distri buted Processing*, pp. 1–8, Malaga, January 1994.

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

[91] L. H. Soicher, "GRAPE: A system for computing with graphs and groups," in *Groups and Computation*, L. Finkelstein and W. M. Kantor, editors, pp. 287–291, 1997. DIMACS Series in Discrete Mathematics and Theoretical Computer Science 11.

[92] A. K. Somani, "Reliability modeling of structured systems: Exploring symmetry in state-space generation," To be presented at the 1997 Pacific Rim International Symposium on Fault-Tolerant Systems, Dec. 14–16, 1997, Taipei, Taiwan., 1997.

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

[94] K. S. Trivedi, B. R. Haverkort, A. Rindos, and V. Mainkar, "Techniques and tools for reliability and performance evaluation: Proble ms and perspectives," in *Computer Performance Evaluation Modelling Tools and Techniques*, G. Haring and G. Kotsis, editors, pp. 1–24, New York, 1994, Springer-Verlag.

[95] J. E. Tvedt, *Matrix Representations and Analytical Solution Methods for Stochastic Activity Networks*, Master's thesis, The University of Arizona, 1990.

[96] B. P. Zeigler, *Theory of Modelling and Simulation*, Wiley, New York, 1976. Reissued by Krieger Publishing Company, Malabar, Florida, 1985.

[97] B. P. Zeigler, *Multifacetted Modeling and Discrete Event Simulation*, Academic Press, London and Orlando, Florida, 1984.

[98] B. P. Zeigler, *Object-Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endormorphic Systems*, Harcourt Brace Jovanovich, New York, 1990.