© Copyright by David Daly, 2001

ANALYSIS OF CONNECTION AS A DECOMPOSITION TECHNIQUE

BY

DAVID DALY

B.S., Syracuse University, 1998

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2001

Urbana, Illinois

ABSTRACT

Realistic computer systems are hard to model using state-based methods because of the large state spaces they require and the likely stiffness of the resulting models (because activities occur at many time scales). One way to address this problem is to decompose a model into submodels, which are solved separately but which exchange results. We call modeling formalisms that support the exchange of results between models "connection formalisms."

This thesis develops connection as a decomposition technique. The existing connection infrastructure in the Möbius modeling framework is used to develop a terminology to describe the decomposition of large models and the inherent associated problems. A theory is then developed to describe when such decompositions are feasible.

The theory is then applied to a special class of models to develop a new set of connection-based approximation techniques that reduce state-space size and solution time. This is done by identifying submodels, called *isolated submodels*, that are not affected by the rest of a model and solving them separately. A result from each solved submodel is then used in the solution of the rest of the model. We demonstrate the use of two of these approximation techniques by modeling a real-world file server in the Möbius modeling framework. The connected models were solved one to two orders of magnitude faster than the original model, with one of the decomposition techniques introducing an error of less than 11%.

To my father, who will never see this thesis, but saw so many other things.

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor William H. Sanders, for technical advice and support on the Möbius project. Jenny Applequist was an invaluable part of the editing and preparation of this thesis, and I would like to thank her for her help and patience. I would also like to thank all the members of the Möbius project past and present, namely Amy Christensen, Graham Clark, Dan Deavours, Salem Derisavi, Jay Doyle, G.P. Kavanaugh, Doug Obal, John Sowder, Aaron Stillman, Alex Williamson, and Patrick Webster, as well as all of the members of the PERFORM research group with whom I have worked.

I would also like to thank Motorola for funding the Motorola Center for High-Availability System Validation at the University of Illinois (under the umbrella of the Motorola Communications Center). I would also like to thank the National Science Foundation (contract number EIA 99-75019), and the Defense Advanced Research Project Agency, Information Technology Office (contract number DABT63-96-C-0069) for their financial support.

Several people have supported me in a nontechnical manner that was invaluable. I would like to thank my wife Orit, my mother Myra, my sister Lisa, and my innumerable friends and family for their support and love through this and everything else.

TABLE OF CONTENTS

CHAPTER PAGE

1 INTRODUCTION . . . 1
- 1.1 Techniques for Modeling Complex Systems . . . 3
  - 1.1.1 Increased levels of abstraction . . . 3
  - 1.1.2 Advances in model construction and solution techniques . . . 4
  - 1.1.3 Connection techniques . . . 7
- 1.2 Example: Computer Architecture Modeling . . . 9
- 1.3 Thesis Contribution . . . 11

2 MODELS AND SOLUTIONS . . . 15
- 2.1 Submodels . . . 16
- 2.2 Solutions of Measures . . . 18
- 2.3 Approximations . . . 20

3 GENERAL CONNECTION . . . 22
- 3.1 Connection as Decomposition . . . 23
- 3.2 Passable Models, Error-Minimizing Models, and Accuracy . . . 26
- 3.3 Example Submodels . . . 28
- 3.4 Types of Connection . . . 31
- 3.5 n-Way Decomposition . . . 35
- 3.6 A Candidate Connection Formalism: FiFiQueues . . . 39

4 MODELS WITH ISOLATED SUBMODELS . . . 43
- 4.1 The Connection Formalism . . . 43
- 4.2 Practical Implications . . . 48

5 CASE STUDY . . . 51
- 5.1 NetApp Filer . . . 51
- 5.2 Model of the Filer . . . 52
- 5.3 Connection Models of Filer . . . 56
- 5.4 Model Measures . . . 59

6 RESULTS . . . 63
- 6.1 Solution Time . . . 64

6.2 Steady State Variable Results 65
6.3 Duration Variable Results 66
6.4 Summary of Results 68

7 CONCLUSIONS 70
7.1 Future Work 71

REFERENCES 73

LIST OF TABLES

Table Page

2.1 Summary of Notation 18

5.1 Gates in Request Submodel 54
5.2 Gates in Filer Submodel 56
5.3 Gates in Driver Submodel 59
5.4 Steady-State Performance Variables 60
5.5 Duration Performance Variables 61

6.1 Solution Times in Seconds 64
6.2 Steady-State Results and *Diff* Values, Bursts of 800 65
6.3 Steady-State Results and *Diff* Values, Bursts of 850 65
6.4 Steady-State Results and *Diff* Values, Bursts of 900 66
6.5 Duration Results and *Diff* Values, Bursts of 800 67
6.6 Duration Results and *Diff* Values, Bursts of 850 67
6.7 Duration Results and *Diff* Values, Bursts of 900 67

LIST OF FIGURES

Figure Page

2.1 Isolated Submodel and Quotient Submodel of a Model 17
2.2 Model with Two Isolated Submodels 17
2.3 Isolated Submodel inside an Isolated Submodel 18

3.1 Generic Connection Model 23
3.2 Two-Queue Network with Feedback 30
3.3 A Cyclic Connected Model with Three Passable Submodels 36
3.4 A Cyclic Connected Model with Three Passable Submodels 36
3.5 Three-Queue Network with Feedback 38
3.6 Submodel of the Three-Queue Network Passing Results Through a Model 39

4.1 Sample Path of the Reconstructed Random Process in a Quotient Submodel When Using the Continuous-Time Random Process Abstraction 44
4.2 Sample Path of the Reconstructed Random Process in a Quotient Submodel When Using the Discrete-Time Abstraction 45
4.3 Sample Path of the Reconstructed Random Process in a Quotient Submodel When Using the Average-Passing Abstraction 47
4.4 Sample Path of the Reconstructed Random Process in a Quotient Submodel When Using the Random-Variable-Passing Abstraction 47

5.1 Model of Markov-Modulated Arrival Process 53
5.2 Model of NetApp Filer Creating a New Snapshot 55
5.3 Driver Submodel for Simplified Model 58

6.1 Distribution of Delays for Complete Snapshot for Bursts of 850 68
6.2 Distribution of Delays for Complete Snapshot for Bursts of 850, Deterministic phase3 Distribution 69

CHAPTER 1

INTRODUCTION

A common difficulty encountered in modeling is the complexity of the systems to be modeled. The modeling tools that are available are overwhelmed by two facets of the complexity: the level of detail required to represent the systems and the number of time scales at which significant events occur in the systems. The high level of detail leads to very large state spaces (and therefore unacceptably large memory requirements) if state-based analytical modeling is used, and it leads to large memory requirements if simulation is used. A model having significant events at many time scales may become stiff, leading to unacceptably long solution times for analytic solutions and simulations.

Complexity is a problem in most modeling domains, including such varied areas as the dependability of fault-tolerant software, performance of CORBA services, performance of MEMs storage devices, microprocessor performance, manufacturing systems, and many other areas. For instance, fault-tolerant software contains several components or modules that interact in complex ways with each other and the environment, as a result of which a large amount of detail is necessary to capture the behavior of the software. In addition, faults occur relatively infrequently, causing the entire system to exhibit behavior on many time scales. In a similar manner, asynchronous communication in CORBA involves several objects interacting through the use of one of a variety of services. The services provide a range of service guarantees, such as guaranteed delivery or message ordering. The performance of the communication services depends on the behavior of a number of objects, as well as the

organization and behavior of the communication service. Burstiness in the communication patterns leads to behavior on many time scales, while the interactions of the objects and the service results in a high level of detail for a model of asynchronous communication in CORBA.

Microelectromechanical systems (MEMS) storage devices may provide a performance and cost improvement compared to magnetic disks. The performance of MEMS devices depends on the physical characteristics of the read heads and of the media used. Factors include the number of heads, the size of the media, and the way that the media or heads are moved, leading to a highly detailed model. MEMS devices are intended to be used in computer systems, and modeling the devices as part of a computer system results in a model with behavior on many time scales, from the cycle time of the processor to the execution time of the test application. The time scales are similar to the time scales in the evaluation of new computer architectures, although the computer architecture may require even more time scales to represent the detailed operation of the processor. The computer architecture has a lot of detail in addition to many time scales, as it contains millions of transistors and uses millions to billions of bytes of memory.

A manufacturing system may produce many types of products, which may be customized according to customer requirements, using many machines and resources. The machines and resources may break or become inaccessible during the manufacturing process. The number of products, the level of customization, the number and arrangement of machines, and the number of resources leads to a detailed system. The availability of machines and resources combined with the requirements of the customers changes at a much slower time scale than is needed to describe the construction of individual products, resulting in a system with significant behavior at multiple time scales.

All of these examples demonstrate complicated systems possibly interacting with complex environments. The examples require models to have a high level of detail and to have behavior at multiple time scales. Capturing all of the detail leads to large models with the

corresponding memory and state-space problems, while the number of time scales leads to stiff models, which are time-consuming to solve.

1.1 Techniques for Modeling Complex Systems

Many advances and developments have led to an increased ability to solve complex models. The advances have come from new ways of representing systems, as well as better methods for calculating measures on models. While these advances greatly increase the scope of systems that can be modeled, there are still limitations that must be addressed. A brief review of some of these advances is now given.

1.1.1 Increased levels of abstraction

One way of dealing with an overwhelming amount of detail is by increasing the level of abstraction in the model, or correspondingly reducing the level of detail. Reducing the level of detail should reduce the size of the state representation and the size of the state space, as well as the number of time scales in the model. If the details that are removed do not affect the considered measures, then no accuracy is lost by their removal. In general, the goal is to minimize the loss of accuracy while decreasing the size of the model.

Several modeling formalisms have been developed to facilitate abstraction by reducing the level of detail or allowing a compact representation of detail. Such formalisms include Markov chains (MCs), Petri nets (PNs), stochastic Petri nets (SPNs) [1, 2], generalized stochastic Petri nets (GSPNs) [3], stochastic activity networks (SANs) [4], deterministic and stochastic Petri nets (DSPNs) [5], queuing networks (QNs), layered queuing networks (LQNs) [6], discrete event system specification (DEVS) [7], VHSIC (very high speed integrated circuits) hardware description language (VHDL), process algebras (PAs), and stochastic process algebras (SPAs). All of these formalisms allow a modeler to describe a system at a high level, or at multiple levels of detail.

DEVS, among other formalisms, allows for explicitly specifying a single entity at multiple levels of abstraction. The multiple levels of abstractions are related through the use of morphisms. A morphism between two systems at one level of abstraction ensures certain morphisms between the two systems at a higher level of abstraction. Based on morphisms, a simpler model may be substituted for and solved instead of a more complex model.

The ability to model a system at a high level provides a good way to develop models that are tractable. However, the trade-off for lowering the amount of detail can be an increase in the error of the model. The job of managing this trade-off is necessarily left in the hands of the modeler. The ability to model a system at multiple levels of abstraction provides extra flexibility in managing the trade-off between the level of detail and error in the model. However, the usefulness of modeling at higher levels of abstraction is limited when the high-level models do not reflect low-level changes to the system. Therefore, increasing the level of abstraction is a good approach to deal with complexity, but is limited by the fact that it might not reflect changes to parts of the system that have been removed from the model.

1.1.2 Advances in model construction and solution techniques

Work has also been done on improving the techniques used to create and solve models. While increasing the level of abstraction removes detail, the model construction and solution techniques attempt to handle the level of detail in the model more efficiently. To deal with the amount of detail of large systems, these techniques attempt either to create smaller, equivalent representations, or to solve larger representations. Stiffness is dealt with mainly by developing techniques to speed up the solution time.

Multiple techniques for use with numerical solutions have been developed to reduce the information needed to represent a system. The techniques focus on exploiting symmetries available in the system to create equivalent but smaller models. They generally work by creating larger models by combining smaller models, which is called *composition*. The replicate/join [8] formalism is a composition technique for SANs that combines models by sharing

state. One way to combine models is to have multiple identical copies of a submodel, which are called *replicates*. The replicate does not explicitly store the state of each submodel, but rather the number of copies of the model that are in a particular state. Storing the number of models in a state, rather than recording which models are in that state, decreases the overall number of states of the entire model.

Symmetry detection has also been used to reduce the size of stochastic well-formed colored Petri nets (SWNs) [9]. SWNs are based upon colored Petri nets, which are an extension of Petri nets. The color extension allows for differentiating types of tokens in the system. An algorithmic technique exists to exploit structural symmetries in the construction of the reachability graph for SWNs. In addition, symmetries in the colors can also be detected and exploited to reduce the state space of the model. Symmetry in the colors is used to identify and remove redundant colors. The removal of redundant colors does not affect the subsequent analysis of the system.

Graph composition [10] is an extension of the symmetry detection of the replicate/join formalism, and it also combines models (not necessarily SAN models) by sharing a portion of the state of each submodel. However, graph composition detects all the symmetries exposed at the composition level and uses them to reduce the underlying state space, instead of requiring the use of a special operation (such as a replicate in the replicate/join formalism). The graph composition uses the detected symmetries to replace each set of equivalent states with one aggregate state, reducing the total state-space size.

Similar work has also been done for stochastic process algebras. Stochastic process algebras are inherently compositional, as expressions are built up from simple statements. The compositional nature of stochastic process algebras is used to systematically replace parts of the model with smaller, equivalent terms based on the notion of bisimulation. This systematic replacement can be done in such a way as to ensure the smallest possible representation (maximum lumping) for the model.

Other work has been done on increasing the size of model that can be solved numerically.

Typically, this work has focused on solving models with large state spaces. Deavours and Sanders [11] developed techniques for generating large state-space representations of models and ways of solving these large state-space representations [12]. Large state spaces were generated by maintaining in memory only a minimal description of each state, instead of the full state descriptions, since information such as the value of the measures on a state is not needed to generate the state space. Large models were solved by generating portions of the state space on the fly, lowering the amount of memory needed to solve the model. Blocks of the state space were generated on the fly and used by a variant of the block Gauss-Seidel iterative solution method.

Models with very large state spaces can be solved using Kronecker operations, provided the infinitesimal generator can be expressed using Kronecker operations [13]. If the generator matrix can be expressed using Kronecker operations, the submatrices that are combined to form the complete generator matrix can be analyzed individually. Solving portions of the generator matrix individually greatly reduces the memory required to solve the complete model, allowing a more efficient solution of the system of equations than could otherwise be had.

Some models with infinite state spaces can be solved numerically, provided that certain restrictions are met. One example of infinite models that can be solved numerically is the general product form solution of an open queuing network. Another example is the application of spectral expansion [14] to the solution of infinite SPNs. Spectral expansion works by identifying collections of states, called *levels*, that repeat infinitely often. The model is solved first for all the levels that do not repeat, in the same manner that a finite model would be solved. Separately, the repeating levels are solved, and the two solutions are combined through a computation of normalizing constants for the two solutions.

The solution of stiff models has been addressed by other recent work. Several techniques have been developed to decrease the time needed to perform numerical solutions. The solution time may be decreased through the use of operations that are usually faster and more

efficient for the solution of a linear system of equations. Stiffness that is a result of simulating systems with rare events has also been addressed. Obal [10] developed importance sampling techniques for SAN-based reward models. Importance sampling is applied by biasing the model such that the rare event becomes more common. The results are then conditioned based on the biasing to reconstruct the unbiased result.

More recent work has been done by Tuffin and Trivedi [15] on importance splitting techniques for SPNs. Importance splitting works by splitting the simulation path into sets. A Bernoulli trial is created to determine whether the next set in the simulation path is hit, and a sequence of conditional probabilities are developed. The probability of the rare event occurring can be computed from the conditional probabilities to get the unconditioned probability of the rare event.

The advances discussed above (creation of smaller equivalent models, solution techniques for large models, and faster solution techniques) greatly increase the level of model detail and stiffness that can be analyzed in a reasonable amount of time. Unfortunately, many models are still too complex and stiff to solve using these techniques. Most construction and solution techniques are limited by the type of models to which they can be applied, and may still be overwhelmed by the complexity of a model even when it is of the appropriate type. For instance, spectral expansion allows the analysis of certain infinite SPNs. However, it is easy for a model to have behavior that causes the levels to become too detailed to analyze, or to have too many transient levels to analyze before the repeating behavior begins.

1.1.3 Connection techniques

A third technique that has been used to model complex and stiff systems is the passing of results between models that are solved separately; we call this procedure "connection." There are many domain-specific examples of models decomposed into submodels that are solved separately and which exchange results. However, the more general techniques that have been developed for decomposing a model into submodels, which are solved separately and

exchange results, are more interesting examples of connection. The more general techniques are applied to QNs, SPNs, or MCs (each of which were discussed in Section 1.1.1), which are very general modeling formalisms. Some of the techniques decompose models based on the time scales of portions of the models, while other techniques decompose models into submodels that have limited interaction with the other submodels.

Time scale decomposition is a technique for studying stiff systems with behavior on very different time scales. Ammar and Islam [16] developed a time scale decomposition technique that can be applied to SPNs. The transitions in the model are partitioned into two sets, a set of fast transitions and a set of slow transitions, based on the assumption that the fast transitions are orders of magnitude faster than the slow transitions. The model is solved for each of the two time scales (fast and slow). The fast model is solved with all of the slow activities removed from the model for each stable marking of the slow model. The results from the fast model are passed to the slow model. The slow model is solved by replacing all the fast transitions with immediate transitions (no delay), and the state-dependent rates are determined by the results of the fast model. The error from using time scale decomposition has been determined to be $O(\epsilon)$, where ϵ is the degree of coupling between the levels. The degree of coupling is related to the ratio of transition rates of the fast and slow transitions. The error is low as long as the difference in rates is large. If there is not a large difference between the rates of the fast and slow transitions, this technique will have a large error, even if the model has many time scales.

Other connection techniques have been developed based on modeling logical units of a system separately. These units may be complex, but interact with the rest of the system in a limited manner. Stochastic rendezvous networks [6] are used to model parallel or distributed systems with synchronization. Stochastic rendezvous networks consist of tasks that take a random amount of time to complete and that may require the services of other tasks in order to complete. Each task is modeled separately, with the dependencies between tasks specified. The tasks are ordered, and solved iteratively. A task can be called by other tasks,

and the analysis of a task uses the results that describe the arrivals of the requests to that task. Similarly, a task can call other tasks, and the analysis of a task also uses the results that describe the waiting time of its requests at other tasks. The results are mean rates and mean waiting times, which are inherently steady-state representations of the system.

Logical units are also used in a connection technique for stochastic reward net models [17]. The connection technique is applied by determining *nearly independent* submodels, where *near independence* means that the underlying continuous-time Markov chain (CTMC) transition matrix has off-diagonal blocks that are almost completely zero, and are the transition matrices for the submodels. If the off-diagonal blocks were completely zero, then the model would be independent, and the submodels could be analyzed in isolation. Even though there is dependence between the models, they are solved separately. The results from the other submodels are used as parameters in the analysis of a submodel. The dependence between the models introduces error into the model.

The connection techniques discussed build upon the use of abstraction and the use of better model construction and solution techniques. The use of the connection techniques allow for the solution of larger and stiffer models than could otherwise be solved. However, the connection techniques often require strong assumptions (just as the time scale decomposition does or as the decomposition techniques for stochastic reward net models do), and may result in large error.

1.2 Example: Computer Architecture Modeling

Computer architecture modeling is an example of a modeling domain for complex systems, and exhibits the problems associated with the modeling of complex systems. Computer systems suffer especially from the amount of detail needed to model the systems and the number of significant time scales in the systems. The high speed of modern processors creates activities that occur at the nanosecond level, but the behavior of the systems at time levels

noticeable by humans, such as seconds, minutes, or hours, is also of interest. That is a range of 9 to 12 orders of magnitude. The fact that these systems have become very complicated is another factor. It is not uncommon to have systems with billions of bytes of memory. The state of such systems is too large to represent in a simulation, and the state space of these systems would be practically infinite.

Many tools are used to model current architectures and their features. There are cache simulators, trace-driven simulators, and execution-driven simulators, all of which try to completely capture the behavior of part or all of a system. However, they are large tools that will have trouble modeling future-generation architectures, which will be much more complex than the architectures currently used to run the tools.

Some of the techniques described in Section 1.1 that are used for modeling large and stiff systems have been applied to computer architecture modeling, while other techniques have not been applied, since the models are too large and stiff to solve even after using those techniques. The most frequently used technique is that of increasing the level of abstraction. The Smart [18] simulation environment studies cache protocols by using an abstract representation of the rest of the computer system. The execution of an application in the system is modeled using the MINT simulator, which is a hybrid simulator that achieves high performance by executing some of the code on the real hardware used to run the simulation, rather than simulating the processor executing the code. The simulation of the application execution is used to generate events for the more detailed simulation of the cache and to determine the performance impact of different cache organizations on application performance. The system simulation is split into the two parts, the detailed cache simulation and the more abstract simulation of the application execution; this is done instead of a detailed simulation of the entire system.

A more abstract representation of the application and processor was used by Stone and Thiebaut [19] in the development of an analytical cache model. Instead of modeling the processor and application in detail, the cache model replaces them with a high-level models

of the memory access pattern. The work is based on the observation that cache misses occur in bursts, mostly due to changes in the working set due either to multiprogramming or to the inability of the complete working set to fit in the cache. Therefore, the specifics of the organization of the processor or the application is of less importance than the memory usage pattern of the application.

Unfortunately, while the abstract models such as the ones described have been accepted, they do not solve the general problem of how to model new computer architectures; instead, they only answer questions focused at particular issues. Other models have also been developed that are more general and use formalisms such as SPN or QN. However, since there are few bounds given on the error of the more general models, the results from these models are not trusted. Instead, detailed simulation is usually used for most modeling problems, despite its problems with scalability and execution space and time.

1.3 Thesis Contribution

The development of the connection techniques discussed in Section 1.1.3 showed promise in addressing large and stiff models (such as the computer system architecture example). However, the techniques discussed are limited in use because of the assumptions they require and the error they introduce. We wish to develop an infrastructure and theory that are useful for analyzing connections for more general systems and that require fewer assumptions. The first work on a connection infrastructure was performed by Christensen [20] as part of the Möbius project [21], and we extend this work by developing a more complete theory of connection.

Earlier, it was simply stated that result passing was connection. A more complete definition of connection would describe it as a form of modeling that decouples the parts of a model. Using connection formalisms, it is possible to solve parts of a model separately to obtain intermediate results that are used in the solution of other parts. Results can be

passed in an arbitrary fashion, possibly in an iterative fashion. By separating a model into submodels, it is possible to reduce the size of the largest state space needed in the solution of the model. If the models are separated so that each submodel has fewer time scales than the original model, each submodel will be less stiff. Ideally, the complete model can be solved using connection in less time than if the complete model were solved as one model.

Connection can be viewed as a decomposition technique for large models, where one large model $\mathcal{M}$ is partitioned into submodels $\{\mathcal{M}_i\}$, with each individual measure $m \in M$ being completely defined on individual submodels in the partition. The submodels are solved separately after the partitioning of the model. Partitioning the model removes all interactions between the submodels. The interaction will be replaced by an exchange of results in the connection model. Therefore, the complete interaction between the two submodels must be captured in the results that are passed between the models. We develop connection as a form of decomposition to be used in the solution of large and stiff models.

Based on that work, we form and address several general questions about connection. What models can be solved using connection? What are the important properties of different types of connection models? And how should abstractions be analyzed? Some notation is developed to express these questions more clearly and to develop answers to these questions. The question of which models can be solved using connection is answered by partitioning all models into four categories: models that can be solved with acyclic connection models, models that can be solved with cyclic connection models for certain initial conditions, models that can be solved as cyclic connection models for all initial conditions, and models that cannot be solved using connection. Classes of models that are contained in each of the given categories are described, and theory is developed to explain why these classes of models are contained in each of the given categories.

The developed theory is then used to analyze a class of models exhibiting a special property. Each submodel in this class has a submodel, called an *isolated submodel*, that is not affected by the rest of the model but that may affect the rest of the model. Such a model

can be decomposed into two submodels: the isolated submodel and a submodel that is the rest of the model (called the *quotient submodel*). We develop a set of connection techniques for use with models that have isolated submodels. These different techniques involve passing a continuous-time random process, a discrete-time random process, a random variable, or an average as the result from the isolated submodel to the quotient submodel.

All of the techniques developed recreate a random process from the abstract result passed, and are therefore called *abstractions*. As would be expected, more assumptions are needed to recreate the random process correctly using the simpler results (random variable and average) than the more complicated results (continuous or discrete-time random process). Of the four techniques, we implement the random-variable-passing abstraction and the average-passing abstraction. The use of the two techniques should result in models that have smaller state spaces and fewer time scales than the original, nondecomposed model, and therefore take less time to solve. However, the improvement in solution time may come with a decrease in accuracy for the model.

After developing these techniques, we demonstrate their use by modeling a real file server. We develop a model of the system and the measures on the model, and then apply the average-passing and random-variable-passing abstractions to the model. We solve the model using discrete event simulation with and without the abstractions, and compare the solution times and results for each solution. The simulation times are one to two orders of magnitude faster for both abstractions, but while the average-passing technique gets good results for some of the measures, the random-variable-passing gets good results for all of the measures we define, with an error below 5% for most of the measures defined, and no higher than 11% for any of the measures in the example.

The rest of the thesis is organized as follows. Chapter 2 introduces the modeling concepts used in our connection theory. The general connection theory and decomposition are developed in Chapter 3. The development of the special class of connected models follows in Chapter 4. Chapter 5 contains a case study that demonstrates the techniques developed

in Chapter 4, and the results of the case study are presented in Chapter 6. We present some concluding remarks and suggestions for future work in Chapter 7.

CHAPTER 2

MODELS AND SOLUTIONS

Central to our work on model connection are the concepts of a "model" and "submodel," since models will be connected together to share results. When connection is used as a decomposition technique, submodels of a large model will be identified so that they can be solved separately and exchange results. We will use the definitions of these terms provided by the Möbius modeling framework [22], and will develop, implement, and demonstrate our ideas using the Möbius framework because of the generality and extensibility it provides. Möbius was developed to be an extensible modeling framework, thus allowing new modeling and solution techniques to be easily added, and has infrastructure developed to support the use of connection.

Without loss of generality to other formalisms and frameworks, we base our definition of a "model" on the definition of an "atomic model" given by Deavours and Sanders [23] for models in the Möbius modeling framework. The parts of this definition that are relevant to our work on connection are presented. In particular, an *atomic model* $\mathcal{M}$ includes the definitions of S, A, and AF, where S is the set of "state variables," A is the set of "actions," and AF is a function called the "action function." A *state variable* is a variable that represents part of the state of the system, and the state of the system is determined by the values of all of the state variables. An *action* changes the value of state variables, and the *action function* determines exactly how each action behaves (e.g., a delay or a state change function). An atomic model is augmented with a set of "measures" M to form a *solvable model*. A *measure*

is a metric defined on the model (concerning its performance or dependability, for example) and can be thought of as a function of a sample path of the model. If it is not clear which model something refers to, we will use a subscript of the model name (e.g., $S_{\mathcal{M}}$ denotes the state variables in $\mathcal{M}$).

2.1 Submodels

Using Deavours and Sanders [23] definition of a model, definitions are provided for a "submodel," an "isolated submodel," and a "quotient submodel." A *submodel* $\mathcal{N} \subseteq \mathcal{M}$ is a model with state variables $S_{\mathcal{N}} \subseteq S_{\mathcal{M}}$, actions $A_{\mathcal{N}} \subseteq A_{\mathcal{M}}$, and AF restricted to the actions $A_{\mathcal{N}}$. No restrictions are placed on the state variables used in the action function. Since there are no restrictions, the behavior of the actions in the submodel (and therefore the behavior of the submodel) may depend on state variables not included in $S_{\mathcal{N}}$, and the actions may change the values of state variables not included in $S_{\mathcal{N}}$.

An *isolated submodel* is a submodel that does not directly affect any of the measures $M_{\mathcal{M}}$ (none of the measures are defined on any of the submodel's state variables or actions) and is not affected by any of the state variables or actions not in the submodel. For instance, the completion of an action elsewhere in the model would not change any of the state variables in the submodel, and none of the actions in the submodel would behave differently if a state variable outside the submodel changed. This situation is illustrated in Figure 2.1. The isolated submodel $\mathcal{N}$ has state variables $S_{\mathcal{N}} \subset S_{\mathcal{M}}$ and actions $A_{\mathcal{N}} \subset A_{\mathcal{M}}$. All the rest of the state variables and actions fall in the submodel $\mathcal{L}$. A certain collection of the state variables $S'_{\mathcal{N}} \subseteq S_{\mathcal{N}}$ and actions $A'_{\mathcal{N}} \subseteq A_{\mathcal{N}}$ in $\mathcal{N}$ affect the rest of the model, which is the submodel $\mathcal{L}$. However, none of the state variables or actions in the rest of the model affect the behavior of $\mathcal{N}$. Also, the measures $M_{\mathcal{M}}$ are only defined on state variables and actions in the submodel $\mathcal{L}$, and not on any of the actions or state variables in $\mathcal{N}$. In general, a model may have more than one isolated submodel (Figure 2.2), and an isolated submodel may itself contain

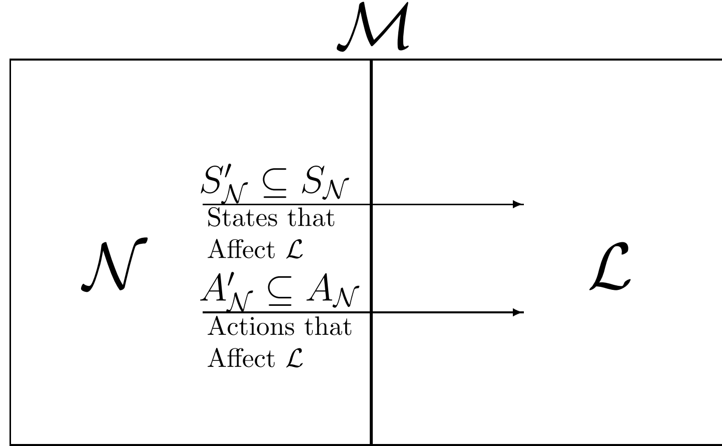

Figure 2.1: Isolated Submodel and Quotient Submodel of a Model

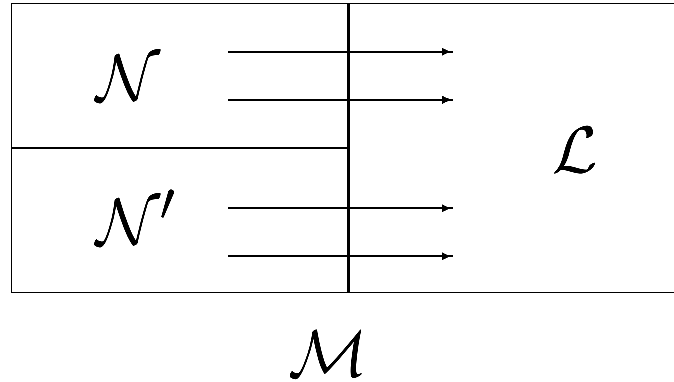

Figure 2.2: Model with Two Isolated Submodels

an isolated submodel (Figure 2.3).

The submodel $\mathcal{L}$ that consists of everything in $\mathcal{M}$ not in the isolated submodel $\mathcal{N}$ is called the *quotient submodel*. In general, its set of measures $M_{\mathcal{L}}$ will be equal to $M_{\mathcal{M}}$, which is the set of measures of the complete model $\mathcal{M}$. $\mathcal{L}$ is not a complete model by itself, since its actions may depend on state variables or actions in $\mathcal{N}$. To be a complete model, $\mathcal{L}$ must be augmented in some way. This augmentation, which consists of passing results to the model, will be provided later.

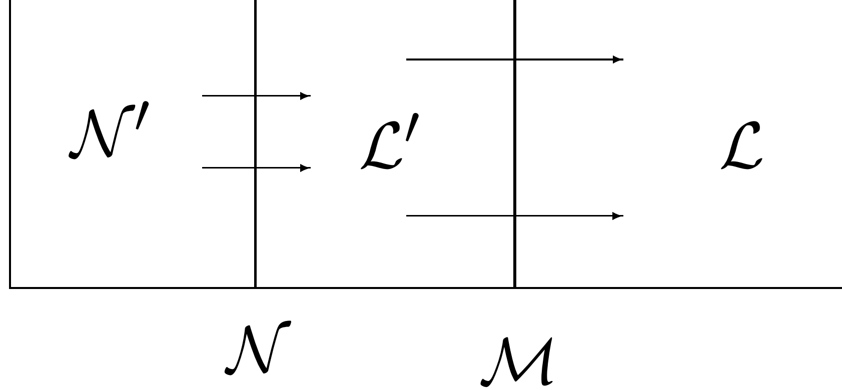

Figure 2.3: Isolated Submodel inside an Isolated Submodel

Table 2.1: Summary of Notation

$S(m)$	A solution of measure m by some solution technique.
$ES(m)$	Exact solution of the measure m.
$Diff(m_1, m_2)$	Generic measure of difference between the solutions of two measures. This is a user-defined function that is nonnegative.

2.2 Solutions of Measures

Some notation is now introduced and is shown in Table 2.1. We define $S(m)$ to be the solution of measure m as determined by some solution technique, and $ES(m)$ to be the exact solution of the measure. Solution techniques implemented in the Möbius tool include discrete event simulation and several numerical techniques. While the numerical techniques can often provide arbitrarily accurate results, we do not consider their solutions to be exact, since in practice the solutions provided by numerical techniques have a small, but usually nonzero, error. We consider a solution to be exact only if there is no error.

The models that are normally studied exhibit stochastic behavior. Therefore, the solution of a measure will be stochastic in nature. A measure can be thought of as a function of a model sample path. If the measure returns one real-valued number per sample path, the solution of the measure will be a random variable, while if the measure function returns a collection of real-valued numbers, the solution of the measure will be a random process. Measures such as the average value of a state variable over a period of time or the value of

a state variable at a specific point in time are examples of measures with random variable solutions. A measure such as the value of a state variable over a period of time is an example of a measure with a random process solution.

The techniques that we are developing are intended to reduce the time and memory required to solve models. Alternate models, which require less time and memory to solve, will be presented that are meant to approximate the original model. We need a way to determine how accurate such models are, and the function $Diff(m_1, m_2) \geq 0$ is defined for this purpose. $Diff$ is a measure of the difference between the solutions of the measures m_1 and m_2, and compares the solutions $S(m_1)$ and $S(m_2)$. The $Diff$ function will be the measure of accuracy for any approximate model generated from a complex model, and will need to be a function of either random variables or random processes, as appropriate, since a solution can be either.

When possible, exact solutions of the measures on the original model and on the approximate model will be used in the evaluation of the $Diff$ function, especially in the development and analysis of new approximation techniques. However, this may not always be possible, in which case we will use the available solutions and note the error of the solution technique.

Possibilities for the $Diff$ function of random variables include (but are not limited to) a function of the probability density function of the random variable, a function of different moments, or the normalized difference in means. Possibilities for the $Diff$ function of random processes include (but are not limited to) a weighted integral of a function of the probability density function of the processes at each point in time, a weighted integral of different moments of the processes at all points in time, or a weighted integral of the absolute difference in means of the processes at each point in time. Other $Diff$ functions for random processes include functions of multiple time points or functions of the covariance functions of the processes, as two processes may have identical behavior at each individual point in time, but still have different behaviors over intervals of time. Combinations of the two given types of functions of random processes can also be used, as well as other functions of random

processes. In general, the *Diff* function will need to be specified by the modeler, to reflect differences in the important characteristics of the random variables with respect to the meanings of the particular performance or dependability variables used.

2.3 Approximations

The idea of a quotient model approximating an original model is now introduced, based on the definition of the *Diff* function. There are three types of approximation: "approximates," "completely approximates," and "approximates to ϵ." A model $\mathcal{M}'$ *approximates a model* $\mathcal{M}$ *($\mathcal{M}' \approx_M \mathcal{M}$) with respect to the submodel* $\mathcal{L}$, if both $\mathcal{M}'$ and $\mathcal{M}$ have $\mathcal{L}$ as a submodel, and $Diff(m_{\mathcal{M}'}, m_{\mathcal{M}}) = 0, \forall m \in M$ defined on $\mathcal{M}$. If this is the case, the solutions of the two models will produce the same answer for all measures defined on $\mathcal{M}$. There is no restriction made on any of the nonmeasured behavior, and the two models could conceivably be quite different. However, those differences are unimportant from the modeler's point of view, as they do not affect any of the measures. The flexibility provided by not restricting the nonmeasured behavior may lead to improvements in model solution time and space.

A model $\mathcal{M}'$ *approximates completely a model* $\mathcal{M}$ *($\mathcal{M}' \approx_{M_\infty} \mathcal{M}$) with respect to the submodel* $\mathcal{L}$, if both $\mathcal{M}'$ and $\mathcal{M}$ have $\mathcal{L}$ as a submodel, and $Diff(m_{\mathcal{M}'}, m_{\mathcal{M}}) = 0, \forall m \in M_\infty$, where M_∞ is the set of all possible measures that can be defined on $\mathcal{L}$. This is a much stronger condition than simply approximating a model. Basically, the two models behave exactly the same over the submodel $\mathcal{L}$, and no differences between them, either unimportant or important, are allowed. This level of approximation allows much less flexibility than normal approximation, and most likely will not lead to improvements in model solution. However, the concept should prove useful in developing other approximations. The set M_∞ can be determined by exploring all possible measure functions $MF(m)$, with the requirement that the model be well-defined [24].[1] Previously, it was mentioned that a measure is a mapping

[1]A model that is not well-defined contains some nondeterminism. A measure could be defined on the portion of the model containing nondeterminism, but the solution of that measure cannot be computed.

from a sample path to a real number. A sample path may be viewed as a mapping from time to the real numbers. Therefore, the set of all possible measure functions can be viewed as the set of mappings from functions from time to the real numbers, to the real numbers $(MF(m) : (T \rightarrow R) \rightarrow R)$.

A model $\mathcal{M}'$ *approximates* $\mathcal{M}$ to ϵ *(*$\mathcal{M}' \approx_\epsilon \mathcal{M}$*) with respect to the submodel* $\mathcal{L}$*,* if both models $\mathcal{M}'$ and $\mathcal{M}$ have $\mathcal{L}$ as a submodel, and $Diff(m_{\mathcal{M}'}, m_{\mathcal{M}}) < \epsilon, \forall m \in M$. This is a looser condition than approximation, as defined earlier. In this case, we are not only allowing non-measured behavior to differ between the models, but are also allowing the measured behavior to differ. However, since the variation is bounded to a small value, that is often a reasonable option. The increased flexibility in constructing the approximate model, compared to the other forms of approximation, should lead to the largest improvements in solution efficiency. We expect that most of the connection techniques we develop will fall into this category.

CHAPTER 3

GENERAL CONNECTION

As described earlier, connection is a technique for solving a collection of models. With connection, results are exchanged between the models in an arbitrary fashion. A *result* is a representation of the solution of a measure, such as a mean or a distribution. Solving the models individually can be more efficient than structurally composing the models together to form one large model, and should be useful in the modeling of systems or collections of systems that are too large or too stiff to model practically using composition. The basic infrastructure for connection in the Möbius framework was developed by Christensen [20] and is built on top of the results database of Möbius.

A generic connected model, as described in [20], is a graph consisting of a set of model nodes called *solvable models*, arcs called *conduits*, and transformation nodes called *connection functions*, as shown in Figure 3.1. A solvable model is a model with a specified reward structure that can be solved; a conduit transfers results; and a connection function takes a set of results to generate an input parameter for another model. The connection graph is traversed in a certain order:

1. A solvable model is solved, the output conduits for that solvable model transfer specific results from the model to a connection function.

2. The connection function performs some mathematical transformation on all the results passed to it.

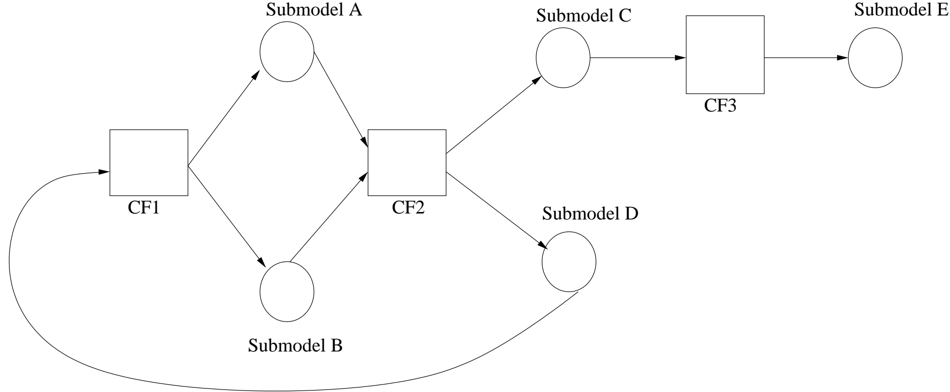

Figure 3.1: Generic Connection Model

3. A conduit provides this value to another solvable model to be used as an input.

4. The process is repeated starting at another solvable model.

This is a very general specification, allowing for a good deal of flexibility in constructing connection models and formalisms. A connection formalism would specify the valid set of connection functions and graph topologies, and would also provide a way to specify the order of the graph traversal and stopping criterion.

3.1 Connection as Decomposition

As described informally in the introduction, connection can be viewed as a decomposition technique for large models, with which one large model $\mathcal{M}$ is partitioned into a set of submodels $\{\mathcal{M}_i\}$, which are then solved separately, with each individual measure $m \in M$ being completely defined on an individual submodel in the partition. Separating the model in this way removes all interactions between the submodels during their execution. This interaction is replaced by exchange of results in the connection model. Therefore, the interaction between the submodels must be captured in the results that are passed between the models.

The submodels interact when some subset of state or action behavior in one submodel

affects the behavior of other submodels. In a state-based model, a submodel can be "affected" by state variables, or by actions in other parts of the model (i.e., another submodel). A state variable *affects* a submodel if any of the actions in the submodel change their delay distribution, enabling, or completion behaviors when the value of the state variable changes. An action *affects* a submodel if the values of any of the state variables in the submodel can change when the action completes.

In order to capture these effects, a new submodel $\mathcal{E}_i$ can be created to represent the effect of the other submodels on each submodel $\mathcal{M}_i$. The results from the solution of the other submodels can be used to parameterize the behavior of this new submodel. A submodel $\mathcal{M}_i^+ = \mathcal{M}_i \cup \mathcal{E}_i$ can then be created that includes the effect of the other submodels on the submodel $\mathcal{M}_i$ through the results passed to it and the submodel $\mathcal{E}_i$. The choices of the structure of $\mathcal{E}_i$ and the results to pass to it determine how "accurate" the submodel will be. If $\mathcal{E}_i = \mathcal{M} \setminus \mathcal{M}_i$ is chosen, then $\mathcal{M}_i^+ = \mathcal{M}$, and the solution of the measures of $\mathcal{M}_i^+$ will be the same as the solution of the same measures on $\mathcal{M}$, but no efficiency in solution has been gained. Therefore, the key to effective use of connection as a decomposition technique is to find models $\mathcal{E}_i$ and results to pass to those models that make it easier to solve $\{\mathcal{M}_i^+\}$ (possibly repeatedly) than to solve $\mathcal{M}$, without much loss of accuracy. We will refer to the set of submodels $\{\mathcal{M}_i\}$ as the *partition of the model* $\mathcal{M}$*,* and we will refer to the set of submodels $\{\mathcal{M}_i^+\}$ as the *decomposition of the model* $\mathcal{M}$*.*

We first consider models that decompose into two submodels $\{\mathcal{M}_1^+, \mathcal{M}_2^+\}$. Since the model is decomposed into two connected models, the order of traversal of the connected graph is clear once one model has been selected to be solved first. After the first model is solved, then the other model would be solved using the results from the first model. The first model would then be solved again, and so on. The only other thing needed to define this connection model completely is the termination condition. One choice is for the model solution to terminate when the change in results over a complete iteration is less than some threshold.

Models $\mathcal{E}_1$ and $\mathcal{E}_2$ generally require results as inputs. These results represent the effects of state variables and actions on the models $\mathcal{M}_1$ and $\mathcal{M}_2$, as just described. There might not be appropriate measures already defined on the other model to pass to this model that represent these effects, in which case extra measures will need to be defined. These measures are defined on the models $\mathcal{M}_1^+$ and $\mathcal{M}_2^+$ as appropriate. In that way, two sets of measures are defined on both $\mathcal{M}_1^+$ and $\mathcal{M}_2^+$. Defined on each model is some subset of the measures M, and extra measures to pass to the other model.

The behavior of an action or a state variable can be captured as a random process. Conceptually, this can be done for a state variable by defining a measure on it at every point (of the possibly uncountably infinite points) in time, with a value equal to the value of the state variable at that particular time. For an action, this can be done by defining a measure for the time of each completion. There would be a measure of the times of the first, second, $\ldots$, n^{th} completions. The exact solutions of these measures would form a random process that captures the details of the effect of a state variable or an action on another submodel. Similarly, the complete behavior of a collection of actions and state variables can be captured in a multidimensional random process. Passing this random process would allow the affected submodel to have a complete description of the elements that affect it.

However, it is not possible to pass a random process as a result; we can only pass results that are some (inherently approximate or incomplete) representation of the random process. Even though a random process cannot be passed in practice, the concept of passing a random process is useful, since it will provide a bound on goodness of techniques that are developed. Random processes as results will be used in the development of the theory of connection. Some types of results that can be passed include mean values, variances, or a distribution function. The choice of the particular result to pass is part of the definition of a connection formalism, and will have a large impact on the accuracy and efficiency of the solution of any model decomposed into a connection model.

3.2 Passable Models, Error-Minimizing Models, and Accuracy

Since decomposing a model into a connection model can introduce error, there is a need for terms to describe this error. We introduce the terms “error,” “accurate,” and “passable” to do this. “Error” and “accurate” are defined on solution of measures, results, and models, while “passable” will only be defined for models. Also used are the definitions of “isolated” and “quotient” submodels presented in Section 2.1. As a reminder, an isolated submodel is a submodel that is not affected at all by the rest of the model, and the quotient submodel is a model with the isolated submodel removed. For the definition of “accuracy” we make use of the definition of the user-defined function *Diff* described earlier for measures whose solutions are random variables or random processes.

Definition 3.1 *The* error of the solution $S(m)$ of measure m *on a connected model* $\mathcal{M}_i^+$ *(that is part of the decomposition of* $\mathcal{M}$*) is* $Diff(m_{\mathcal{M}_i^+}, m_{\mathcal{M}})$.

The error is the measurable difference between solving the decomposed model using the representation of the process that affects it, and solving the model using the actual process. It assumes that the solution of m on the main model is an exact solution and has no error itself, due to solution technique. The definition of “error” is dependent on the specific definition of *Diff* provided by the modeler. This definition is also applied to the extra measures defined on $\mathcal{M}_i^+$, by adding those measures to the measures defined on $\mathcal{M}$.

Definition 3.2 *The* solution of a measure is accurate *if its error is zero.*

Therefore, a measure is accurate if there is no difference between the values obtained from solving the decomposed model and the values obtained from solving the original model.

Similarly, we define the terms *error* and *accurate* for a result. The information lost by the particular representation of the solution of the measures is not considered. Instead, the

accuracy and error of a result depend only on the error and accuracy of the solution that produces the result.

Definition 3.3 *The* error of a result *of a connected model $\mathcal{M}_i^+$ is the error of the measure solution on which the result is based.*

Definition 3.4 *A* result is accurate *if its error is zero.*

The definitions of "error" and "accurate" for a model follow from the definitions for measures.

Definition 3.5 *The* error of a solution of a connected model $\mathcal{M}_i^+$ *is the greatest error of all the solutions of the measures defined on the model. The measures defined on the model include a subset of the measures M in $\mathcal{M}$, and the extra measures passed to other models.*

The error of the solution of the model (which is part of the decomposition of the model $\mathcal{M}$) is the largest error of any of the solutions of the measures, and the solution of any measure has an error no more than the error of the model. Ideally, models will have very small or no error.

Definition 3.6 *The* solution of a connected model $\mathcal{M}_i^+$ is accurate *if its error is zero.*

Ideally, at the termination of the solution of the connection model, all of the solutions of the connected models will be accurate, and the model solution will have taken less time than solving the model without decomposing it would have. This goal will usually be unattainable due to approximations that are used, and some trade-off in accuracy versus solution time will need to be chosen.

The definitions of "passable model" and "error-minimizing model" are based on the definitions for "accurate." We have discussed how a model can be decomposed into two models, with certain results passed between the models. The results are from the solution of the measures defined on the state variables and actions in one model that affect the other model. These measures can also be defined on the nondecomposed model.

Definition 3.7 *A* connected model $\mathcal{M}_i^+$ is passable with respect to a result *if it can be solved for an accurate solution when it is passed that result and that result is accurate.*

The result passed to the connected model represents the actions and state variables in other models that affect this model. In order for a model to be passable, the result passed to it must capture all of the needed detail from the other model, and this detail must be used to properly reconstruct the process that affects this model. Therefore, for a model to be passable with respect to a result, it must be possible to reconstruct the process that affects it based on that result, in such a way that the model has an accurate solution.

Definition 3.8 *A* connected model $\mathcal{M}_i^+$ is error-minimizing with respect to a result *if the error of the solution of the model is less than the error of the result passed to it for that result.*

It is clear that a model that is error-minimizing with respect to a result is also a passable model with respect to that result, since if it has an accurate result passed to it, the solution of its measures must also be accurate. It is not necessary that a model that is passable with respect to a result also be error-minimizing with respect to that result.

3.3 Example Submodels

Based on the definition of a passable submodel, submodels that are and are not passable are discussed, as well as submodels that are error-minimizing. Unless a submodel is an isolated submodel, it is affected in some way by the rest of the model. As discussed earlier, results representing those effects can be passed to the submodel. The submodel uses the result to recreate the effect of the rest of the model on the submodel and that result can be used in any way to recreate the interaction between the submodel and the rest of the model.

A submodel has some effect on the other submodel, unless that other submodel is an isolated submodel. The effect should be reflected in the result passed to the first submodel,

creating feedback. It might be the case that the feedback cannot be reflected properly in the result to be passed. For example, consider a situation in which (a) an activity completes probabilistically based on the value of a state variable that is not in the submodel, and (b) the firing of the activity modifies the value of the state variable. When the action completes, the state variable should change, but the random process recreated from the result will not change and cannot capture the interaction. This could lead to a measurable difference in model execution in some cases. A submodel exhibiting such behavior would not be a passable submodel.

With that in mind, some models are now shown to be passable.

Theorem 3.1 *An isolated submodel is a passable submodel with respect to all results.*

An isolated submodel does not depend on anything outside of itself, so it will get an accurate solution regardless of the result passed to it. □

This is a simple, but useful result. Also, it is clearly true that an isolated submodel is an error-minimizing model, since it can always be solved accurately.

Theorem 3.2 *A quotient submodel is passable with respect to a result that completely characterizes the random process of affecting actions and state variables.*

There is no feedback loop between the submodels, since the quotient submodel has no effect on the isolated submodel. Therefore, there is no difference between solving the quotient submodel by itself with the random process recreated from the result passed to it, and solving the quotient submodel as part of the complete, nondecomposed model. □

Therefore, the class of models that have an isolated submodel is a subset of the set of models that can be decomposed into passable models. It is irrelevant (although not meaningless) whether a quotient submodel is error-minimizing with respect to any result, since its results are not used by any other submodels.

Theorem 3.3 *A model containing an isolated submodel can be decomposed into two passable submodels.*

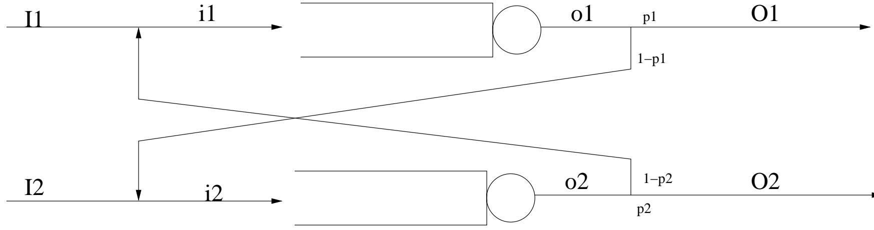

Figure 3.2: Two-Queue Network with Feedback

The isolated submodel and the quotient submodel are both passable submodels, by the previous two theorems. Therefore, decomposing a model into its isolated and quotient submodels will also decompose the model into passable submodels. □

We note that the three theorems that show that isolated and quotient models are passable do not depend on the definition of the measures on the original model, even though the definition of "passable" explicitly depends on the accuracy of the measures. This is a result of the structure of a model with an isolated submodel, and need not be the case for other kinds of models.

A simple example of a model that has passable submodels, but does not include an isolated submodel, is an open stable queuing system with feedback, shown in Figure 3.2. While this system can be solved easily without connection, it is still a useful example of connection. Jackson's theorem shows that the steady-state distributions of the lengths of the queues are mutually independent, and dependent on the mean arrival rate to each queue, provided that arrivals into the system are Poisson. Therefore, if the measures on each queue are taken to be the steady-state queue length distribution and the mean departure rate, each queue forms a passable submodel with respect to the mean departure rate of the other submodel. The mean departure rate of each queue is passed to the other queue. The mean rate is used to determine the rate of a Poisson arrival process for the other queue. Given the correct mean from the other model, the model will get accurate results for departure rate and state occupancy, and is therefore passable with respect to the mean departure rate.

These submodels are also error-minimizing submodels with respect to the mean departure

rate, provided that $p1 < 1$ and $p2 < 1$. If either of the two models is passed an incorrect rate (with a difference of e), then its departure rate will have an error, but the error in that rate will be $p1$ (or $p2$) $\times\, e$. Therefore, on every solution iteration of the model, the error of the submodels will decrease, and each submodel is an error-minimizing model. This is true for any Jackson network of queues with feedback.

3.4 Types of Connection

Viewing connection as a binary decomposition technique, we can divide models into four broad classes with regard to connection:

1. Models that are solvable as acyclic connected models.

2. Models that are solvable as cyclic connected models for some set of initial conditions.

3. Models that are solvable as cyclic connected models for all initial conditions.

4. The models that are not solvable using connection.

Trivial connection models, such as one-node connection models or models with $\mathcal{E}_i = \mathcal{M} \setminus \mathcal{M}_i$, are not considered. Also, the efficiency of a decomposition is not considered. A future question is what models fall into each class when "solvable" is replaced with "solvable efficiently" in the above list.

In the discussion of model solutions, we assume that the result passed is the complete random process of affecting elements, even though the complete process cannot be used in practice. Since all other types of results that are passed attempt to recreate the process, our assumption will give us a fundamental bound of what is solvable using connection. The following sets of models are contained in each of the classes of models given. Each set of models is contained in the corresponding class:

1. The set of models that have an isolated submodel.

2. The set of models that do not have an isolated submodel, but are decomposable into passable models.

3. The set of models that do not have an isolated submodel, but are decomposable into error-minimizing submodels in which the error decreases fast enough to converge to zero.

4. The set of models that are not decomposable into passable models.

We show that the sets of models are contained in the corresponding classes of models in steps. Models with isolated submodels are dealt with first.

Theorem 3.4 *The set of models that have an isolated submodel is a subset of the set of models that can be solved accurately with an acyclic connected model.*

Given a model containing an isolated submodel, the isolated submodel can be solved accurately by itself (by Theorem 3.1). The accurate result can then be passed to the quotient model. Since the quotient model is passable with respect to the random process of affecting elements (by Theorem 3.2), it will yield an accurate result when passed an accurate result. Therefore, the entire model can be solved accurately with an acyclic connected model. □

Theorem 3.5 *The set of models that can be decomposed into passable submodels (with respect to the results we consider) is a subset of the set of models that can be solved accurately by using a cyclic connected model for some nonempty set of initial conditions.*

Given a model that is decomposable into passable submodels, if one of the submodels is solved accurately, it will pass an accurate result to the other submodel. The other submodel will be solved accurately, and the solutions will remain fixed and accurate after that point, since the submodels are passable. Therefore, the correct results form a fixed point; if the solution reaches that point, it will terminate there with the correct solution. Since the accurate solution forms one possible set of initial values, there is at least one initial value for which the solution of the model will converge. □

Theorem 3.6 *The set of models that can be decomposed into error-minimizing submodels (with respect to the results we consider) in which the error decreases fast enough to converge to zero is a subset of the set of models that can be solved accurately with a cyclic connected model for all initial conditions.*

Given a model that can be decomposed into error-minimizing submodels, every iteration through the solution of each submodel will result in a decrease in error, since the submodels are error-minimizing. When each submodel is solved, its result will have a lower error than the result that was passed to it. The solution of the other model must have an error that is lower than the result passed to it, which is lower than its previous solution. If the rate at which the error decreases is high enough (such as decreasing on average by a constant factor),[1] the error will converge to zero, causing the model to be solved accurately. Since a model that is error-minimizing with respect to a result is also passable with respect to that result, the solution will stay at the correct value once that value is achieved. Therefore, given a model that is decomposed into error-minimizing submodels, if the rate of decrease for the error drives the error to zero, the model can be solved accurately with a cyclic connected model for all initial conditions. □

Theorem 3.7 *The set of models that cannot be decomposed into passable submodels is a subset of the set of models that cannot be solved accurately with a connected model.*

Given a model that cannot be decomposed into passable submodels, at least one of the submodels is not passable. The nonpassable submodel will not yield an accurate result when passed an accurate result. Therefore, even when the other submodel gets an accurate result, this submodel will not. Its inaccurate result will either yield a wrong answer for a measure on the model, or force the other submodel to get an inaccurate result. For this reason, a model that cannot be decomposed into passable models cannot be accurately solved using either cyclic or acyclic connection. □

[1]Conceivably there could be a sequence of error values $e_i = 1 - \sum_{j=1}^{i} 10^j$. While the error value is strictly decreasing, it does not go to zero.

The error introduced by the inaccuracy may be acceptable. The error might not propagate and cause the solution to diverge, but rather lead to only a small error in the final result. If the inaccuracy leads to a small error, we may still want to solve the model using a cyclic connection, even though it cannot get an accurate solution. This concept can be captured by loosening the definition of "passable" submodel to become " ϵ-passable" by replacing the term "accurate" with "ϵ-accurate." A result is *ϵ-accurate* if $Diff(m_1, m_2) < \epsilon$, where m_2 is the corresponding measure defined on the non-decomposed model.

Theorem 3.8 *The set of models that can be decomposed into ϵ-passable submodels is a subset of the set of models that can be solved ϵ-accurately with a cyclic connected model, for some nonempty set of initial conditions.*

Given a model that is decomposable into ϵ-passable submodels, if one of the submodels is solved ϵ-accurately, it will pass an ϵ-accurate result to the other submodel. The other submodel will be solved ϵ-accurately, and the solutions will remain in the ϵ-neighborhood of the accurate solution, since the submodels are ϵ-passable. Provided that the solution of the submodels is ϵ-accurate on some iteration, when the solution of the cyclic model terminates, the solutions will still be ϵ-accurate. Since all values that are ϵ-accurate are possible sets of initial values, there is at least one initial value for which the solution of the model will terminate with an ϵ-accurate result. □

The value of ϵ need not be the same for every submodel, as long as the error introduced by a model for a result is less than the error that can be tolerated by the models that consume that result. This technicality can be removed through an appropriate scaling of the error values.

It is also worth noting that a model that is decomposable into ϵ-passable submodels is not guaranteed to converge, even if the solution becomes ϵ-accurate. The solution of the measures may oscillate within the ϵ-neighborhood of the accurate solution and never satisfy the convergence criteria. However, it is true that if the solution terminates, if the solution

of the models was ever ϵ-accurate, the solution of the models will be ϵ-accurate when the solution terminates.

3.5 n-Way Decomposition

New issues arise when a model is decomposed into n submodels. The main complication comes when multiple models pass results to a single model $\mathcal{M}_i^+$. If the model depends on multiple models, it will get results from all of the models. These results represent the behavior of the actions and state variables that affect the model. Unfortunately, there is no way to determine the joint probabilities of the random processes that the results represent after the model is decomposed. In the binary case, that was not an issue, since all the random processes came from a single model, and the result passed represented one multi-dimensional random process.

Two ways of addressing the problem are presented. Both address the problem for a Three-way decomposition, since the extension of the Three-way case to an arbitrary n-way decomposition is straightforward. Consider a model $\mathcal{M}$ that can be partitioned into three submodels $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$, where each of the submodels affects the other two submodels. The model can be solved in two ways. The first way, shown in Figure 3.3, is simply to assume that the random processes that the results represent are actually independent. This is similar to what is done in the analysis of Jackson networks, and if this assumption is made, it is easy to recreate the random process of affecting state variables and actions. However, the assumption may not be true; the second manner of solving the model addresses that case.

If the random processes represented by the results are not known to be independent, their joint behavior should be reflected in the reconstructed processes. Figure 3.4 shows a model that reflects the joint behavior of the processes passed in Figure 3.3. The model is a connected model in which each submodel receives only one result. It is important that the

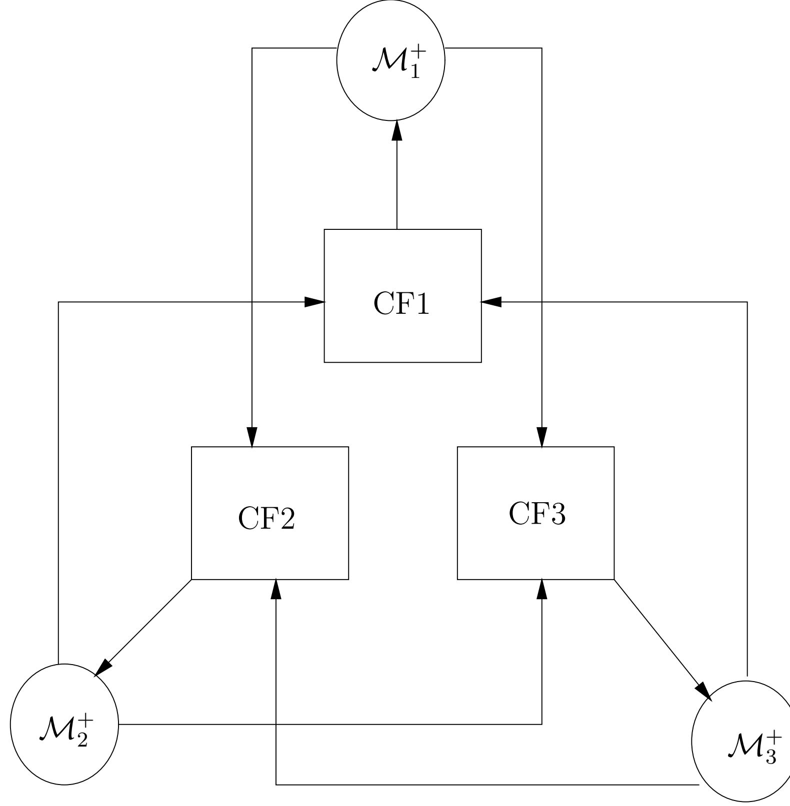

Figure 3.3: A Cyclic Connected Model with Three Passable Submodels

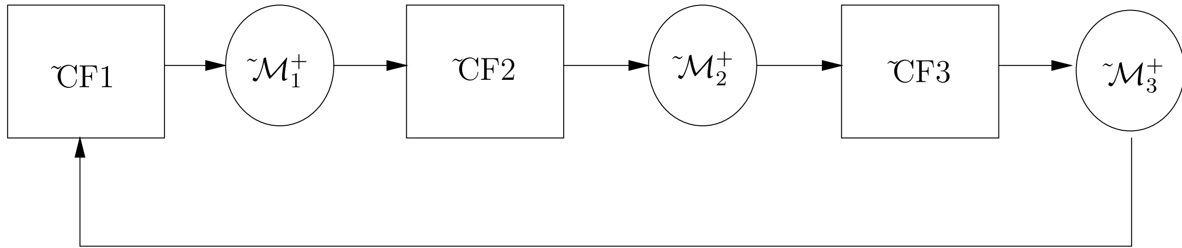

Figure 3.4: A Cyclic Connected Model with Three Passable Submodels

one result contain all the information needed by the submodel to recreate the processes that affect it. In the figure, $\tilde{\mathcal{M}}_2^+$ passes a result to $\tilde{\mathcal{M}}_3^+$. However, $\tilde{\mathcal{M}}_3^+$ also needs to know the effect of $\tilde{\mathcal{M}}_1^+$ on it. This is accomplished by defining more measures on $\tilde{\mathcal{M}}_2^+$. The added measures are defined on the result passed into $\tilde{\mathcal{M}}_2^+$ from $\tilde{\mathcal{M}}_1^+$. Because of the definition of these measures, the effect of the random process from $\tilde{\mathcal{M}}_1^+$ is included in the result passed to $\tilde{\mathcal{M}}_2^+$ and passes through $\tilde{\mathcal{M}}_2^+$ to be included in the result passed to $\tilde{\mathcal{M}}_3^+$. Because the result is passed through the model, the execution of the model contains the random processes of

affecting elements from this model and the previous model. Since both random processes exist in this model, we can determine the joint behavior between the two random processes, provided that the submodel is still passable with respect to these new inputs and measures. In this way, we now have the joint probabilities without having to solve the larger models.

Passing results through models does require that more information be passed between models, and could slow the convergence of the model solution. In order for a submodel to receive an accurate result, not only does its predecessor node need to get an accurate solution for the measures that affect the submodel, but the predecessor node must also get an accurate solution for the measure on the random process recreated from the result passed into the node. However, if the model is passable, once the model receives an accurate result, the submodel will get an accurate result; and so will the other models, if they are passable.

Passing the results through the models necessitates a stronger requirement for solution convergence. Not only must each submodel be passable with respect to the variables defined on it from the original model and the measures defined on it for other submodels, but it must also be passable with respect to the results being passed through the model. In effect, there are extra measures on the model and extra inputs. The model needs to be passable after the addition of these extra results being passed in and out of the model.

The technique of passing results through a model can be illustrated using the three-node queuing network (QN) shown in Figure 3.5. This QN has Poisson arrivals and exponential service time, and departures are routed to the other queues or out of the system independently and randomly. The model, like the one in Figure 3.2, can be solved for steady-state queue distribution using a product form solution based on Jackson's theorem. However, the product form solution is only useful for measuring steady state distribution of queue lengths and mean departure rates. The product form solution can not be used to solve measures such as distribution of interdeparture times.

The model can be decomposed into three submodels, each of one queue. Figure 3.6 shows how one submodel could be constructed, including the passing of results through the model.

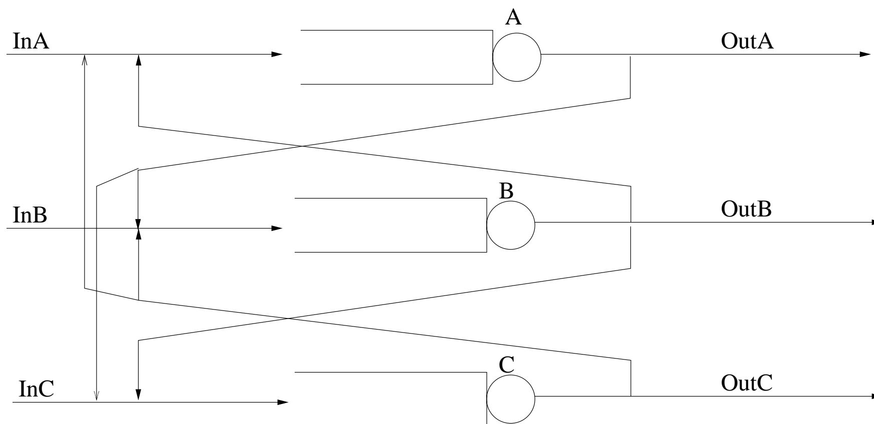

Figure 3.5: Three-Queue Network with Feedback

The model receives one result, which represents the total traffic not leaving the system from the previous queue and the traffic from the last queue (the previous queue's predecessor) that is routed to the queue. The traffic to the queue then consists of some part of the traffic from the previous queue, the traffic from the last queue, and the input traffic. Since the input traffic is Poisson, it will be independent of the other traffic streams, but the two streams of arrivals from the other queues need not be independent. The joint nature of the streams is reflected in the result passed and, therefore, can be recreated properly. Similarly, the joint behavior of the output of the queue and the departures from the previous queue destined for the next queue can be determined from the model, since both are modeled in the model. Therefore, the connection model will reflect more of the time-varying behavior of the three-queue system than the product form solution could. While it reflects more time-varying behavior than the product form, it will not be passable for all possible measures, since there is some interaction between the arrivals and departures of each queue that is not reflected in the model. For example, a burst of departures from the queue could increase the probability of more arrivals in the future, and the increase will not be reflected.

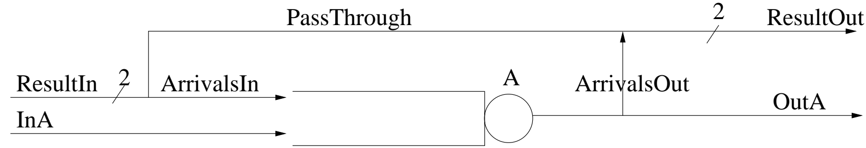

Figure 3.6: Submodel of the Three-Queue Network Passing Results Through a Model

3.6 A Candidate Connection Formalism: FiFiQueues

FiFiQueues [25, 26] is a modeling tool that has been developed at Rheinisch-Westfälische Technische Hochschule to model a class of open queuing networks. The queuing stations can have finite- or infinite-size buffers, and are general phase-type queues, where the arrival process and service times are phase type distributions. The output of each queue is probabilistically routed to other queuing stations or out of the system.

Each queuing station is solved separately using an exact analysis of the CTMC, even for an infinite buffer size queue. Infinite buffer size queues are solved using spectral expansion [14], an exact numerical solution technique. The queuing stations are solved iteratively, with results passed between the queues on each iteration. The solution terminates when the difference in results from one iteration to the next is below some threshold.

The phase-type (PH) departure streams are passed between models, as represented by the mean and squared coefficient of variation of the streams. This representation is an approximation, since phase-type distributions are not completely described by their first two moments. The output stream from a queuing station is probabilistically split to feed other queuing stations or to leave the queuing network. The splitting of the output process is performed on the representation of the stream and is therefore a function of the mean and squared coefficient of variance of the output stream. The calculated values for mean and squared coefficient of variance for the split stream are exact, provided that the output process is a renewal process. If the output process is not a renewal process, the splitting is approximate for the two values.

The arrival process for a queue consists of a combination of split departure processes and

arrivals from outside the queuing network. These processes are combined before analysis of the node to form the arrival process for the queue. The streams are combined according to the assumption that they are independent, using an approximate superposition algorithm that is a function of the mean and squared coefficient of variance of all of the input streams. The PH streams could be exactly combined into a large PH stream through a simple combination of all of the PH types (provided the processes were independent), but doing so would result in a large state space to represent the arrival process. The approximate superposition algorithm that is used should not introduce significant error while leading to simpler calculations than those involved in the use of the exact PH stream.

FiFiQueues is a more general model than the one (examined in Section 3.3) of the two-node queuing network with feedback, Poisson arrivals, and exponential service time, since it has general phase-type service distributions. FiFiQueues can be described using connection terminology, with which each queuing station can be viewed as a solvable model that is a submodel of the queuing network; the means and squared coefficients of variance are the results that are passed; and the splitting and superposition of the streams based on the results passed are performed by connection functions. There are three possible sources of error for FiFiQueues; these may cause the submodels to be not passable with respect to the means and squared coefficients of variance passed for certain measures defined on the model. The first source of error is representing the streams by the mean and squared coefficient of variation. Further error is introduced when the streams are merged at the input of a node. The merging of the streams assumes that the streams are independent, which may not be true; furthermore, an approximate technique is used to merge the streams based on the assumption. The final source of error is the loss of interaction between the input and output of the queuing stations. The output of a queue is fed to other queues, and in part is returned to the queue. There may be some joint time-varying behavior between the input and output that cannot be captured in the model, since the output cannot directly affect the input. So even though each queue can be solved accurately in isolation for a given arrival

process, it cannot necessarily be solved accurately as part of the decomposition of the larger queuing network.

The fact that these queuing networks cannot be decomposed into submodels that are passable with respect to all measures is not a large limitation. The use of approximation will generally lead to submodels that are not passable. In the list of sources of errors, most of the sources were approximations that could be replaced with error-free representations at the cost of a more expensive solution. A similar model could pass the exact PH streams as results by passing the CTMC description of the streams and could combine the streams exactly in the connection functions. If the output streams are independent and the interaction between input and output can be captured, the submodels would form passable submodels with respect to the exact PH streams. FiFiQueues serves as a feasible, approximate version of the model. Therefore, under slight restrictions, the system is decomposable into passable submodels and is therefore solvable using connection for some set of initial conditions.

The analysis of FiFiQueues involves a n-way decomposition. The merging of the input streams in FiFiQueues addresses the same problem as combining the results in a connection model. Since FiFiQueues assumes that the arrival processes are independent, its method of combining results is equivalent to the first method presented in the n-way decomposition section (Section 3.5), which also assumed that the processes represented by the results were independent. It has not been established that the assumption is valid. If the assumption is not valid, using the second method of passing results through models to solve for their joint behavior described in the n-way decomposition section could lead to more accurate results for a wider range of measures.

In this section, we have described FiFiQueues using the language developed for connection by providing a mapping to solvable models, results, and connection functions from FiFiQueues. An analysis of FiFiQueues using the theory of connection showed when an idealized version of FiFiQueues could be used to generate passable submodels that could be iteratively solved for an accurate result. FiFiQueues's use of the approximation in the stream

superposition introduces error. An interesting question arises from expressing FiFiQueues in the connection language: is the approximate model ϵ-passable? If we could determine when it is, that would lead to a better understanding of when FiFiQueues will converge to the proper answer, and an understanding of ϵ would lead to a better understanding of the fundamental accuracy of the model.

CHAPTER 4

MODELS WITH ISOLATED SUBMODELS

A model with an isolated submodel is a special case of the theory developed in the Chapter 3. It was shown in the previous chapter that a model with an isolated submodel can be decomposed into an acyclic connected model, which can be solved accurately. To obtain an accurate result, it is necessary to pass a random process of the affecting actions and state variables from the isolated submodel to the quotient submodel, but it is not usually possible to do that. Instead, the result that is passed will usually be some approximation of the process. The class of models that contain isolated submodels is a good class of models to use in testing such approximations.

4.1 The Connection Formalism

We have developed a set of abstractions to be used on acyclic connected models. These abstractions exploit the fact that the model has an isolated submodel, and they were developed independent of any particular solution technique and the constraints imposed by a particular solution technique. The focus in developing these abstractions was on the trade-offs between accuracy and the information that needs to be passed between models. Later in this section, the feasibility of the abstractions is analyzed with regard to known solution techniques. The derived connected model allows for the solution of the isolated submodel

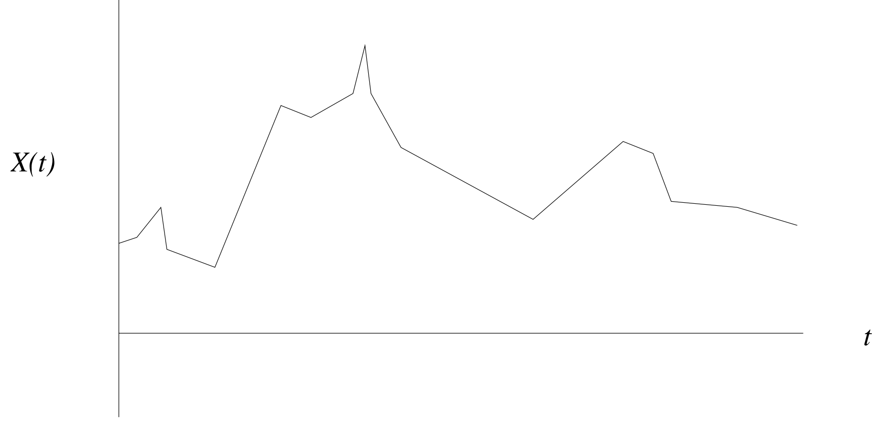

Figure 4.1: Sample Path of the Reconstructed Random Process in a Quotient Submodel When Using the Continuous-Time Random Process Abstraction

$\mathcal{N}^+$ and quotient submodel $\mathcal{L}^+$, with results being passed between the two models, rather than the direct solution of $\mathcal{M}$, where $\mathcal{N}^+$ and $\mathcal{L}^+$ are the decomposition of $\mathcal{M}$. Both models $\mathcal{N}^+$ and $\mathcal{L}^+$ have some extra features for use in the connection that were not in the submodels $\mathcal{N}$ and $\mathcal{L}$ that are the partition of $\mathcal{M}$. The model $\mathcal{N}^+$ can be solved by itself, since it does not depend on anything in $\mathcal{L}^+$. It is not clear, however, how to solve the model $\mathcal{L}^+$ by itself.

To capture all the behavior of $\mathcal{L}$ in $\mathcal{L}^+$ as it would appear as part of $\mathcal{M}$, the complete effect of the isolated submodel $\mathcal{N}$ on $\mathcal{L}$ needs to be included in $\mathcal{L}^+$. Section 3.1 describes how measures (M') could be defined to capture the complete effect of affecting elements on a submodel. The exact solutions $ES(M')$ (Figure 4.1) of these measures defined on $\mathcal{N}^+$ for $\mathcal{L}^+$ would form a (possibly multidimensional) random process that captures the complete details of the effect of $\mathcal{N}$ on $\mathcal{L}$. Since it was shown in Theorem 3.2 that a quotient submodel is passable with respect to the random process of affecting elements, if the random process $ES(M')$ was passed from $\mathcal{N}^+$ to $\mathcal{L}^+$ and used in $\mathcal{L}^+$ to replace the effect of $\mathcal{N}$, we would obtain exactly the same behavior in $\mathcal{L}$ whether it is solved in $\mathcal{L}^+$, or as part of $\mathcal{M}$. So, as described in Chapter 2, the model $\mathcal{L}^+$ approximates completely $\mathcal{M}$. Passing the result of the exact solution of the measures is called the *continuous-time random process abstraction.*

Another possibility would be to pass a discrete-time version of the continuous-time ran-

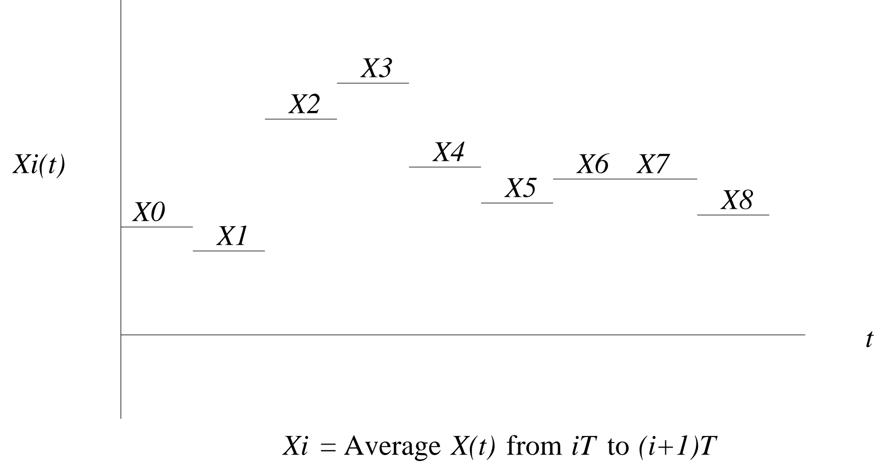

Figure 4.2: Sample Path of the Reconstructed Random Process in a Quotient Submodel When Using the Discrete-Time Abstraction

dom process. The discrete-time version (shown in Figure 4.2) could be created by replacing the measures M' in the previous abstractions with measures M'_T, defined as the time-average values of the state variables and the time-average number of action completions over particular intervals of time, with each interval length being T. If the model $\mathcal{L}$ was insensitive to variations in $ES(M')$ that last less than T time units, the model $\mathcal{L}^+$ would get the same answer for all the measures M defined on the original model $\mathcal{M}$. For example, a model may be affected by an impulse of value a, but only by the increase of $\frac{a}{T}$ in the average value for that interval, and not by the sudden increase and decrease in value. We define the *time sensitivity* T_0 *of* L to be the largest value T such that $\mathcal{L}^+$ is passable with respect to $ES(M'_T)$, as defined in Section 3.2, when it is solved with the discrete-time random process $ES(M'_T)$. The model $\mathcal{L}^+$ will get the same results as the model $\mathcal{M}$ when solved with the random process $ES(M'_T)$ if $T \leq T_0$, or, as described in Chapter 2, the model $\mathcal{L}^+$ will approximate the model $\mathcal{M}$.

The quotient submodel $\mathcal{L}^+$ may be mostly, but not completely, insensitive to variations of $ES(M')$ over periods of time T. Although using this value of T would introduce some error, it might be small enough to warrant solving $\mathcal{L}^+$ with $ES(M'_T)$. We define the ϵ-time sensitivity T_ϵ to be the smallest T such that $\mathcal{L}^+$ is ϵ-passable with respect to $ES(M'_T)$, as

defined in Section 3.2. If $\mathcal{L}^+$ is ϵ-passable with respect to $ES(M'_T)$, the decomposed model will approximate to ϵ the model $\mathcal{M}$ as defined in Chapter 2, which means that the model will get results acceptably close to those of the original model $\mathcal{M}$ when solved with the random process $ES(M'_T)$ and $T \leq T_\epsilon$. Passing the result of exact solutions of the modified measures (whether using T_0 or T_ϵ) is called the *discrete-time abstraction*.

If $T_0 = \infty$, there would be only one interval to solve for and, therefore, only one random variable to pass using the discrete-time abstraction. If the process were ergodic, then the variance of the one random variable would be zero, since the time-averaged mean converges to the ensemble mean in ergodic processes, and every model trajectory would result in the same value for the measure. In that case, passing the average values would yield a model that gets the same result as the original model $\mathcal{M}$, and therefore approximates the model $\mathcal{M}$. Likewise, if $T_\epsilon = \infty$, if the average values were passed, that would still result in a model that approximates to ϵ the model $\mathcal{M}$ and gets results that are acceptably close to those of the original model $\mathcal{M}$. We call this third abstraction the *average-passing abstraction*; the reconstructed process is shown in Figure 4.3. If the model were not ergodic, then the time-averaged value would not converge to the ensemble mean. Different model trajectories would result in different values for the measure, and a random variable would be needed to represent the possible values of the measure.

If $T_\epsilon < \infty$, then passing just the average values would lead to a loss in accuracy. However, if the random variables representing the different time intervals of $ES(M'_T)$ were independent and identically distributed (IID), a simpler form of the random process $ES(M_T)$ could be passed. In that case, the behavior of $ES(M'_T)$ could be completely captured by one measure m' defined on any of the time intervals. The solution of the measure $ES(m')$ would be a random variable that could be passed and used to reconstruct the random process in $\mathcal{L}^+$. We would specify the reconstructed process as $P[i] = ES(m')$, $\forall i$, which would be equivalent to $ES(M'_T)$. This fourth abstraction is called the *random-variable-passing abstraction*.

Even if the process was not completely IID, the random-variable-passing abstraction

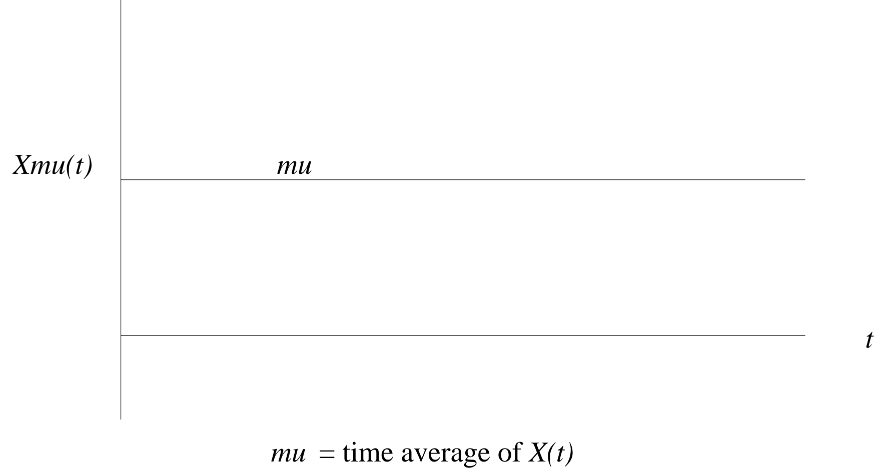

Figure 4.3: Sample Path of the Reconstructed Random Process in a Quotient Submodel When Using the Average-Passing Abstraction

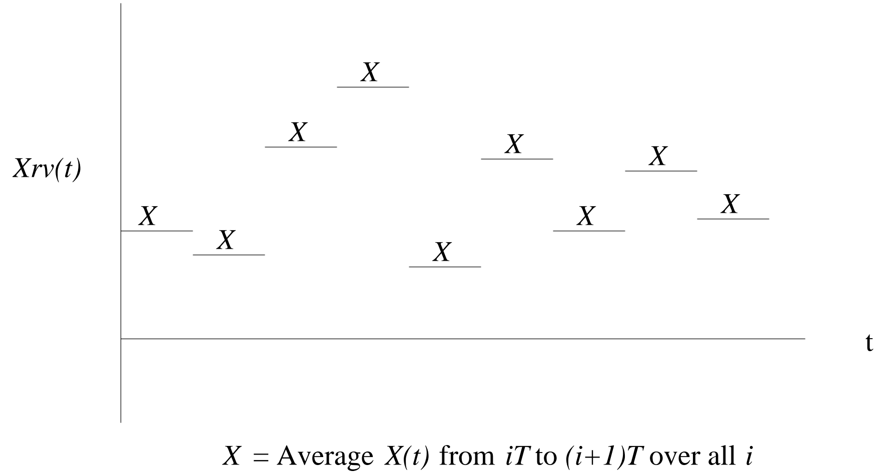

Figure 4.4: Sample Path of the Reconstructed Random Process in a Quotient Submodel When Using the Random-Variable-Passing Abstraction

could be applied if the correlation between intervals was low, and the process was stationary. We define $\bar{T}_0$ to be the smallest value T such that values in different time periods would be uncorrelated, and $\bar{T}_\epsilon$ to be the smallest value T such that the error due to the correlation between distinct time periods would be less than ϵ. The model $\mathcal{L}^+$ with the random variable $ES(m')$ passed to it would approximate the model $\mathcal{M}$, if $\bar{T}_0 \leq T \leq T_0$. The model $\mathcal{L}^+$ with the random variable $ES(m')$ passed to it would approximate to ϵ the model $\mathcal{M}$, if

$\bar{T}_{\epsilon_1} < T < T_{\epsilon_2}$. The relationship between ϵ, ϵ_1, and ϵ_2 would be dependent upon the definition of the $Diff$ function.

4.2 Practical Implications

Solution techniques and practical implications were not considered during the development of the set of abstractions to use in passing results. However, the goal of the work is to find usable abstractions, so the practical implications of the abstractions are now reviewed. The first thing of note is that an arbitrary model cannot be solved for the continuous-time random process defined in Section 4.1. If the type of the random process is known, the model can be solved for the random process's parameters, but the type of the process is not normally known. Similarly, the process can be assumed to be of a particular, general type of random process, and the model could be solved for the parameters for that type of process. Some encouraging work has been performed on this topic, including work by Sadre and Haverkort [27] on using Markovian arrival processes (MAPs) as the general process type in queuing networks. However, without knowing the type of the process or assuming it to be of a certain type, one could naively attempt to solve for the complete process by solving for a finite subset of the uncountably infinite collection of measures; however, solving for all the joint distributions of the measures would be impractical. Therefore, while useful in developing other abstractions, the continuous-time random process passing abstraction can be difficult to use in practice.

We are also unable to solve an arbitrary model for the discrete-time random process described in the previous section. As with the continuous-time random process, if the type of the discrete process is known or is assumed, the parameters of that process can be solved. The exact type of the process is usually not known, but the use of a general process type could be useful in the solution of the processes. Also as with the continuous-time random process, one could naively attempt to solve the complete process without knowing or assuming the

type of the random process. While the number of measures needed to completely solve the discrete-time random process is countable, it is still infinite, and the naive attempt to solve for all the measures would once again fail. Solving the model for a finite subset of the measures (e.g., if we are only interested in solving $\mathcal{M}$ for its measures at some time t) would still be difficult, since the model would still need to be solved for all the joint probability distributions of the measures at all the time periods used. Therefore, while the discrete-time random process passing abstraction is useful in developing the remaining abstractions, it also can be difficult to use in the solution of actual models.

A model can be solved for a random variable in Möbius. Möbius does this by determining the probability distribution of the random variable, which completely describes the random variable. If the model is solved using discrete-event simulation, Möbius develops the distribution by determining what fraction of the sampled values were within each step of the distribution. For a numerical solution, the underlying Markov state space is generated, and the state occupancy probabilities are solved. The value of the measure is determined on each state; this value, together with the state occupancy probabilities, can be used to compute the probability distribution function.

A multidimensional random variable can in theory be solved in a similar manner, although this ability has not yet been implemented in Möbius. The lone difference for a multidimensional random variable is that multiple values need to be compared in the construction of the distribution. For example, consider a sample from a discrete-event simulation, where the sample consists of two values. The two-dimensional step containing the pair of values should reflect the probability mass of the sample. The probability mass will not appear in two separate intervals in two different distributions, as would be the case if there were two one-dimensional random variables, instead of one multidimensional random variable.

A model can also be solved for a mean value of a random variable. The mean value is solved via a method similar to the one described for the solution of a random variable using the state-based numerical techniques, determining the measure value for each state and the

state occupancy probabilities. In discrete-event simulation, techniques such as batch means can be used to compute the average value. These techniques only require the saving of one or a few values, such as the running average, instead of the larger number of values needed for the solution of a random variable. Further, it is possible to solve for the mean value of a multidimensional random variable by solving for the mean value of multiple one-dimensional random variables. This is clear because there is no joint behavior to capture with mean values. Therefore, when mean values are the results being passed, there is no need to assemble the multi-dimensional random variable. Instead, each of the elements can have its own measure, with no loss of accuracy.

CHAPTER 5

CASE STUDY

An example is presented to demonstrate the use of the abstractions developed for models with isolated submodels in Chapter 4 and the abstractions' applicability, accuracy, and performance. Since the last two abstractions (passing a random variable and passing an average) are both feasible, they are both applied to the developed model. The example model needs to exhibit the properties (e.g., containing an isolated submodel) required by the two abstractions; this chapter first describes this model and then shows how it does indeed exhibit the necessary properties. All of the developed models are implemented in the Möbius modeling tool [28, 22] and solved with the Möbius discrete event simulator [29].

5.1 NetApp Filer

A file server from the Network Appliance line of filers was chosen for the study. These filers provide NFS and Windows file services [30], and have demonstrated availabilities of up to 99.99% in production use [31]. In addition, they provide such services as online access to multiple backups called "snapshots" and a way to make consistent off-line backups of the system.

The NetApp filers provide this functionality using a novel architecture [32] that is specifically designed to be a file server. When a filer receives a write request, it goes through three basic steps: it updates its in-memory cache, it logs the request in nonvolatile RAM

(NVRAM), and it replies. The request is not written to disk until up to 10 s have passed, at which time all outstanding accesses are written to disk in one consistent step, generally in under 1 s. The architecture does this by writing all the changed file blocks and updated inodes to unused blocks. When that is done, it writes a new root node, which points to the new directory structure.

In that way, the on-disk file image is always consistent, and older consistent images are also accessible on disk, since they are not overwritten. This mechanism allows for the creation of consistent backups by copying snapshot images from the disk. The backup is consistent because the snapshot does not change. Normally, there is a chance that the files will be updated during a backup, unless operations are suspended.

For this architecture, we want to know what fraction of time the system spends updating the disk image of the file system, and the fraction of time spent in each stage of these updates. We also would like to know how much time each of these updates, and the stages that compose these updates, takes to execute, each time it executes. We will develop a model specifically to answer questions of this type.

5.2 Model of the Filer

The first model developed to answer these questions was the *full model*. The full model does not employ any of the abstractions described and was used to test the quality of the connected models developed. The full model contains an arrival process called the *request submodel*, which represents requests arriving and filling up the NVRAM, and a process called the *filer submodel*, which models the update of the disk. Using Möbius, we defined these two submodels and composed them to make the full model.

The request submodel represents the arrival of requests to the file server and is shown in Figure 5.1. It has two main states, bursty and slow, represented by a token in either the place Bursty or Slow. The rate of the activity Arrival is exponentially distributed, and

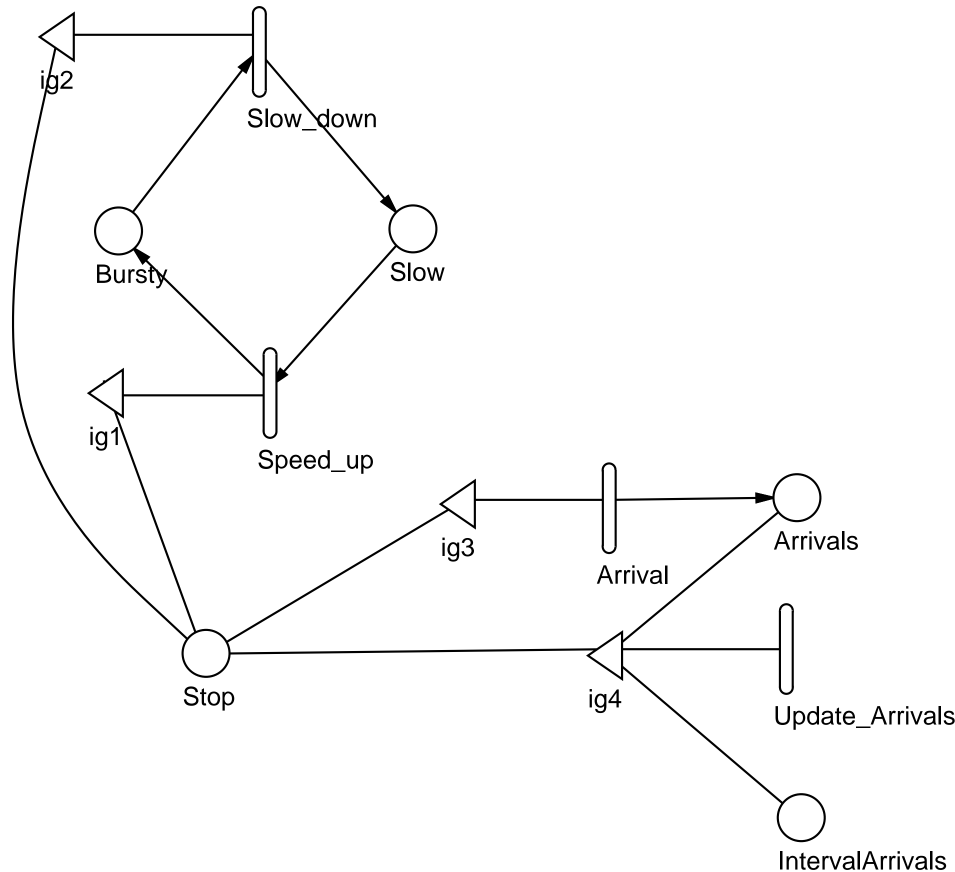

Figure 5.1: Model of Markov-Modulated Arrival Process

its rate is dependent on whether it is in the bursty or slow state, making this submodel a Markov-modulated Poisson process. This model was chosen to represent the bursty nature of requests to a file server. However, the model is just an estimate of the arrival process. Actual request traffic and behavior of the file server users were not used to develop this model; such study would lead to a more accurate model of the arrival process.

For all of the experiments, the low rate was 50 arrivals per second, while the bursty rate was varied between 800 and 900 arrivals per second. The slow state had an average holding time of 4.5, while the bursty state had an average holding time of 1. These numbers were selected so that the average number of arrivals per 10 s would be at about the level needed to cause a disk update to clear the NVRAM. The model should have interesting behavior in this operating region, with a significant number of updates being started by timeouts, and

Table 5.1: Gates in Request Submodel

Gate	*Predicate*	*Function*
Ig1	Stop−>Mark != 0	;
Ig2	Stop−>Mark != 0	;
Ig3	Stop−>Mark != 0	;
Ig4	Stop−>Mark != 0	;

a significant number being started by the filling up of the NVRAM.

The place Stop is used to disable this model. If there is a token in Stop, none of the activities are enabled. In some cases, this provides an improvement in solution time. All submodels composed with the filer submodel will have this feature. This functionality is implemented through the use of *gates*. There are two types of gates: input and output. An output gate only has a function that is called when the action to which the gate is connected is fired. An input gate has that function and also a predicate. The predicate must be true for the action to which the gate is connected to be enabled. The gate functions are written in C++. The gate specifications for the request submodel are shown in Table 5.1. Each of these gates serve to disable an activity if there is a token in Stop. If there is not a token in Stop, they have no impact on the execution of the model.

The filer submodel represents the disk update process and is shown in Figure 5.2. When either the arrivals fill up the NVRAM bank (represented by the place Arrivals), or the timeout occurs (represented by the deterministically timed activity Timeout), the arrival queue is processed. When that happens, the contents of Arrivals is copied to Processed_Arrivals, and Arrivals is cleared. As mentioned in the file server description, the filer has several phases through which it progresses to update the disk. Specifically, there are five phases. First, the filer marks all of its dirty file cache entries as being IN_SNAPSHOT (phase1). It then allocates disk space for all the dirty buffers, and updates the inodes (phase2). This step copies all the inodes to a buffer, and then clears the IN_SNAPSHOT flag for each inode. At that point, most requests can continue. The system then updates its map of in-use disk blocks (phase3), before writing all the dirty disk blocks to disk (phase4). Once all

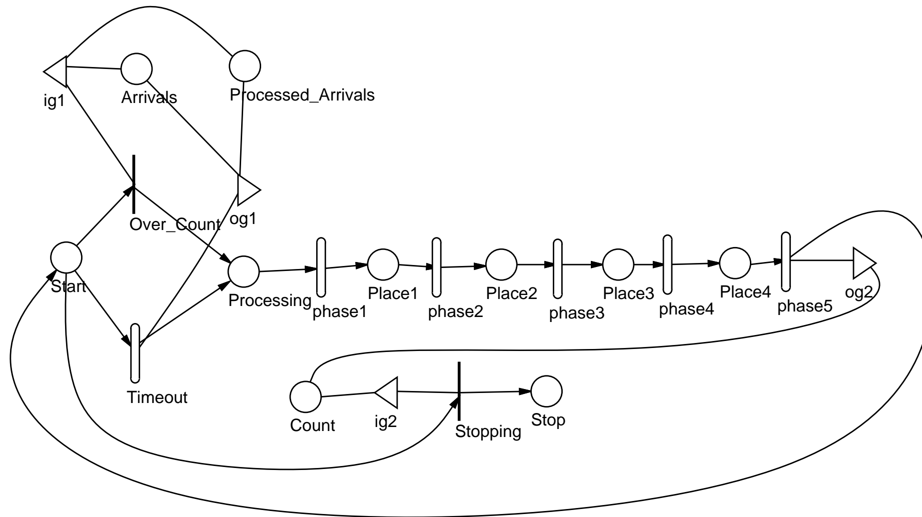

Figure 5.2: Model of NetApp Filer Creating a New Snapshot

the disk blocks have been written, the system copies the root inode and writes it to disk (phase5). Phase1, phase2, and phase4 are modeled as exponentially distributed activities to reflect service time variability due to file size, whereas phase3 and phase5 are modeled as deterministic distributed activities with a constant delay. Phase4 is the write to disk and is the dominant phase with regard to delay.

The gate specifications for the filer submodel are shown in Table 5.2. These gates do more than just disable the model. Ig1 controls the action Over_Count, enabling it when there are more than Threshold arrivals in the queue. When that happens, a disk update is started, the tokens in Arrivals are copied to Processed_Arrivals, Arrivals is cleared, and Full is set to one. Similarly, 0g1 is called when a timeout occurs, in which case the tokens in Arrivals are also copied to Processed_Arrivals, and Arrivals is cleared. Unlike in Ig1, however, Full is set to zero. Ig2 enables Stopping if there is more than one token in Count. This drains the token from Start, disabling the model. Finally, Og2 puts a token in Count every time phase5 completes, if the flag Accumulate_Count is true. The use of the flag Accumulate_Count allowed us to use the count feature only when we needed it.

Table 5.2: Gates in Filer Submodel

Gate	*Predicate*	*Function*
Ig1	Arrivals−>Mark() > Threshold	Processed_Arrivals−>Mark() = Arrivals−>Mark(); Arrivals−>Mark() = 0; full−>Mark() = 1;
Ig2	Count−>Mark() > 1	;
Og1	N/A	Processed_Arrivals−>Mark() = Arrivals−>Mark(); Arrivals−>Mark() = 0; full−>Mark() = 0;
Og2	N/A	if (Accumulate_Count) Count−>Mark()++;

5.3 Connection Models of Filer

The arrival process is an isolated submodel (see Section 2.1) of the full model, since it is not affected by any of the state variables or actions in the rest of the model. Therefore, the developed abstractions can be applied to the full model. For each of the abstractions, a representation of the arrival process needs to be passed from the isolated submodel (request submodel) to the quotient submodel (filer submodel). The abstractions that represent the arrival process with a random variable and with a mean are used.

However, before applying these abstractions, we define *Diff* in order to compare the different abstractions. The *Diff* function is a measure of how accurate the approximate models are compared to the original. In particular, we let *Diff* be the absolute value of the percent error on the means, as shown here:

$$Diff(m_{\mathcal{M}}, m^{+}_{\mathcal{L}}) = \left| \frac{E[S(m_{\mathcal{M}})] - E[S(m^{+}_{\mathcal{L}})]}{E[S(m_{\mathcal{M}})]} \right| \quad (5.1)$$

It will always be expressed in percent format ($XX\%$), rather than in a decimal format ($0.XX$).

To apply the average-passing abstraction, it is necessary to definea measure on the arrivals in the request submodel to determine the average rate of arrivals. For the abstraction to be

applicable, the quotient submodel needs to have $T_\epsilon = \infty$ as its time sensitivity to the arrival process.

In order to apply the random-variable-passing abstraction, a period T is needed such that $T < T_\epsilon$, the time sensitivity of the quotient submodel. If $T \leq T_\epsilon < \infty$, and $T > \bar{T}_\epsilon$ (see Section 4.1), this abstraction should give better results than the average-passing technique. Reviewing the model, we note that the disk updates every 10, or when the queue fills up. We choose $T = 1$ for the period. Compared to the 10 of the timeout, and the time it takes the queue to fill up, this second should be less than T_ϵ for any of the fraction-of-time measures or holding-time measures described in the system description. We also believe that $\bar{T}_\epsilon \leq 1$, since the holding time of the bursty state is 1.

The activity *update_load* is used when the request submodel is solved by itself for the results needed for the two abstractions. Every time the interval of time defined by *interval* has passed, update_load copies all the tokens from Arrivals to IntervalArrivals, making IntervalArrivals represent the number of arrivals in the past interval. Changing the length of the interval effectively changes the period T. A reward variable is defined on the IntervalArrivals place and solved for its distribution of values (for random variable passing) and its mean (for average passing) using the same model solution for both abstractions.

A small model called the *driver submodel* (shown in Figure 5.3) is built to re-create the random arrival process, as required for the random-variable-passing abstraction. The driver submodel is the extra model $\mathcal{E}_i$ used to construct $\mathcal{M}_i^+$ in Section 3.1. Every time *interval* time units pass, it picks another value from the distribution of the random variable. Unfortunately, we do not currently have a general way to sample from a distribution in our simulator implementation. Therefore, an activity with cases was used to mimic the distribution (a case is chosen at random when the activity fires). For the experiments, 10 cases were used to represent the 0th to 10th, 10th to 20th, ... , and 90th to 100th percentiles. The cases use the values of the 5th, 15th, ... , and 95th percentiles to represent these probability intervals, which are all equally probable.

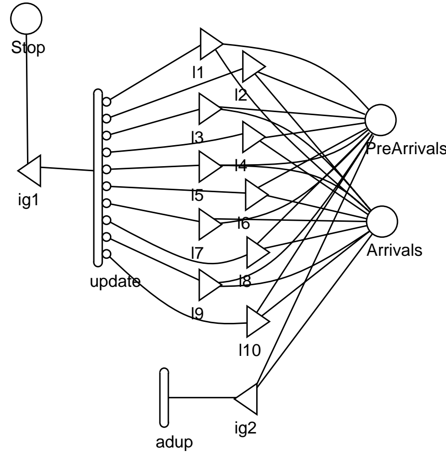

Figure 5.3: Driver Submodel for Simplified Model

The requests are first put in the place PreArrival by the output gate selected by the action. Each output gate represents one of the probability intervals. After 1 s the requests are moved to the place Arrivals, which is shared with the filer submodel. A problem arises if the number of requests in this batch overflows the queue. In reality, the system update would occur before the next complete second completed. The action adup is used to represent this early update. If the queue can only accommodate 1/3 of the current batch (tokens in Arrival), then adup copies 1/3 of those requests to Arrivals2 after 1/3 of a simulated second, triggering a system update.

The gate specifications for the driver submodel are given in Table 5.3. Ig1 is used to disable the update activity if there is a token in Stop. This effectively disables the entire submodel. Ig2 enables adup, if there are more than Threshold tokens in PreArrivals and Arrivals. When adup fires, the gate moves exactly enough tokens to have Threshold + 1 tokens in Arrivals, causing an update. The output gates L1, L1, ... , and L10 are very similar in behavior. When update fires, exactly one of them is called. Each places a set number of tokens in PreArrivals after moving all the tokens from PreArrivals to Arrivals. Since these gates are so similar, we only place one entry in the table for them.

The driver submodel configured in that way (using 10 values) and composed with the filer submodel forms the *random-variable-passing connection model*. The driver submodel is

Table 5.3: Gates in Driver Submodel

Gate	*Predicate*	*Function*
Ig1	Stop−>Mark() == 0	;
Ig2	(Arrivals−>Mark() + PreArrivals−>Mark() > Threshold) && (Arrivals−>Mark() < Threshold)	PreArrivals−>Mark() -= (Threshold + 1 - Arrivals−>Mark()); Arrivals−>Mark() = Threshold + 1;
Li	N/A	Arrivals−>Mark() += PreArrivals−>Mark(); PreArrivals−>Mark() = leveli;

also used to implement the average-passing connection model. It does so by being composed with the filer submodel and setting one case's probability to 1. When that case is selected, the average number of arrivals is placed in Arrivals. The driver submodel configured in that way and composed with the filer submodel forms the *average-passing connection model*.

5.4 Model Measures

These models were developed to determine the fraction of time that the file server spends in certain sets of states in the update process and the holding times in these states. The states of particular interest include the states in which the system cannot process new requests, the update state, and any state that is possibly less fault-tolerant than the rest.

The first two phases of processing in the system block the processing of many requests. Therefore, the model will be considered to be in the blocked state if it is in phase1 or phase2. The last phase is the most critical to correctness, as it is during this phase that the system goes from one consistent disk image to another. An error in this phase could lead to corrupt data; therefore, it is considered the risky state. The model is in the update state if it is in any of the update phases. This is of interest because the whole update process is system-intensive, and therefore degrades the response time of any requests that arrive during that period.

The fraction of total time that is spent in those states, and the holding times of those states (except for the risk state, since it has a known constant duration) need to be measured.

Table 5.4: Steady-State Performance Variables

Proc Time	if (filer->Processing->Mark() + filer->Place1->Mark() +
	filer->Place2->Mark() + filer->Place3->Mark() +
	filer->Place4->Mark() > 0)
	        return (1);
	else
	        return (0);

Arrivals	return(filer->Arrivals->Mark());
Arrivals2	return(filer->Processed_Arrivals->Mark());
Full	return(filer->full->Mark());

Blocking	if (filer->Place1->Mark() + filer->Processing->Mark() > 0)
	        return (1);
	else
	        return (0);

High Risk	return(filer->Place4->Mark());

The fraction-of-time measures are defined as steady-state performance variables in Möbius, and are shown in Table 5.4, while the duration measures are defined as transient performance variables in Möbius, and are shown in Table 5.5. The Möbius reward variable formalism uses C++ syntax to define performance variables as functions, which then return values. The Mark() method is called on a place to determine the number of tokens in the place.

Proc_Time, Blocking, and High Risk are indicator random variables having a value of 1 if a certain condition is true or 0 if it is not. The condition from Proc_Time is the system being in the update state, while the condition for Blocking is the system being in the blocking state, and the system being in the risky state is the condition for the variable High Risk. The average value of the variables is the fraction of time spent in these states, or the probability of being in those states.

Several other steady-state measures are also defined. Arrivals is defined to be the value of the Arrivals place, while Arrivals2 is defined to be the value of the Processed_Arrivals place, and Full is defined to be the value of the place Full. The place Full is set to 1 whenever the filling up of the queue causes a snapshot, and is cleared whenever the timeout causes a snapshot.

Table 5.5: Duration Performance Variables

Wait Time	if (race−>Count−>Mark() == 1 && race−>Processing −>Mark() + race−>Place1−>Mark() + race−>Place2−>Mark() + race−>Place3−>Mark() + race−>Place4−>Mark() > 0)
	    return (1);
	else
	    return (0);

Blocking	if (race−>Count−>Mark() ==1 && race−>Place1−>Mark() + race−>Processing−>Mark() > 0)
	    return (1);
	else
	    return (0);

Finally, we define a set of measures on the holding times of the update state and the blocking state. We do not define a measure on the holding time of the high-risk state, since by the definition of the model, that holding time is constant. These measures are defined as transient performance variables in Möbius, and are shown in Table 5.5. Wait Time is the length of time spent in the waiting state during one occurrence of the waiting state, and Blocking is the length of time spent in the blocking state during one occurrence of the blocking state. For that reason we call these variables the *duration* performance variables.

The intent is for these variables to measure the holding times of these states in steady-state conditions. Unfortunately, we are unable to do that directly. Instead, we measure the holding time of the states during the nth execution of the disk update. If n is large enough, that will be accurate, but the larger n is, the more inefficient the simulation becomes. For our simulations we used $n = 2$, since we believe that the system should settle down to steady-state behavior quickly. We believe this because the filer model saves no state from one update to the next, and the arrival process only stays in the bursty and slow states for 1 and 4.5, respectively, on average, so the arrival process should be in steady state after one update.

The duration variables show the effect that these states have on requests that occur in them. The shorter the durations, the lower the performance impact on such requests.

The steady-state variables, defined on the fraction of time in those states, represent the probabilities that the filer will be in those states. The lower the value of those variables, the lower the probability that a request will be impacted at all by those stages.

CHAPTER 6

RESULTS

The full model, random-variable-passing model, average-passing model, and request model developed in the case study chapter were solved using discrete event simulation for a confidence interval of $\pm 1\%$ with a probability of 95%. While there is no reason why analytical solution techniques cannot be used with the connection techniques used, the nature of the models themselves prevents the use of analytical techniques, since each model has more than one activity with a deterministic delay distribution.

The request model was solved for the number of arrivals per simulated second to provide parameters for the random-variable-passing and average-passing models, while the full model, random-variable-passing model, and average-passing model were solved for the duration variables and steady-state variables described in the previous chapter. All of these models were solved with three different values for the rate of arrivals in the bursty state (or three different sets of results to represent these three different arrival rates). These values were 800, 850, and 900 arrivals per simulated second, which placed the model in an operating region in which updates sometimes occur because of timeouts, and sometimes occur because the disk buffer is full.

Table 6.1: Solution Times in Seconds

Bursty Rate	*Request Model*	*Avg. Pass*	*Avg. Total*	*R.V. Pass*	*R.V. Total*	*Full*
800 Dur.	241.74	34.55	276.29	39.29	281.03	2468.08
850 Dur.	268.14	34.33	302.47	39.48	307.62	2536.67
900 Dur.	280.69	34.36	315.05	39.18	319.87	2477.51
800 Steady	241.74	78.98	320.72	80.52	322.26	6658.02
850 Steady	268.14	80.18	348.32	81.21	349.35	6865.78
900 Steady	280.69	80.95	361.64	82.17	362.86	7178.21

6.1 Solution Time

The times required to solve these models are given in Table 6.1. Since the average-passing and random-variable-passing models require the solution of the request model, we also present the solution time for these models plus the solution time of the request model in the Average Total and R.V. Total columns. However, the same solution of the request model was used to create the results for the average model and random-variable-passing model, for both the duration and steady-state solutions, so each request model solution is thus used at least four times in this example. Therefore, the Average Total and R.V. Total columns are conservative measures of complete solution time of the connected models, while the Avg. Pass and R.V. Pass columns are optimistic measures of the complete solution time of the connected models. Depending on the extent to which the solution of the isolated submodel (the request model) is required, the average time to completely solve such a connected model will vary between the optimistic and conservative values.

Both abstractions were simulated in considerably less time than the full model. With the request model solution time included, the abstractions ran about one order of magnitude faster than the full model; with the request model solution time excluded, it ran almost two orders of magnitude faster. This makes sense for the simulation times that do not include the request model solution time, since in the full model there are activities firing hundreds of times per simulated second, whereas in the average-passing and random-variable-passing models, activities fire no more than once per simulated second. The request model also takes

more than an order of magnitude less time to solve than the complete model. While it has activities that fire hundreds of times per simulated second, just as the full model does, it has fewer time scales, as its slowest activities fire on the order of once per simulated second, while the full model's slowest activities fire on the order of once per ten simulated seconds.

6.2 Steady State Variable Results

The results for the steady-state variables for the three models, and the *Diff* of these results, are presented in Table 6.2 (Bursty rate = 800), Table 6.3 (Bursty Rate = 850), and Table 6.4 (Bursty rate = 900). The random-variable-passing model did well on all the variables in all cases, with *Diff* up to 11% (on the variable Full), while the average-passing model performed well only on the Proc_Time and Blocking variables in all cases. Of the other variables, the best result for the average-passing model was an error of 14.8% on the Arrivals variable for a Bursty rate of 900; its worst result had an error of 83.5% for the Full variable with a Bursty rate of 900. The random-variable-passing model had a much better

Table 6.2: Steady-State Results and *Diff* Values, Bursts of 800

Variable	*Full Model value (m_1)*	*RV Passing value (m_2)*	*Avg. Passing value (m_3)*	*Diff* (m_1, m_2)	*Diff* (m_1, m_3)
Proc Time	$0.1015 \pm 7\times10^{-4}$	$0.1002 \pm 6\times10^{-4}$	$0.1001 \pm 6\times10^{-4}$	0.2%	1.3%
Arrivals	764 ± 3	779 ± 2	912 ± 2	1.9%	19.3%
Arrivals2	1603 ± 5	1712 ± 4	1936 ± 4	6.8%	20.7%
Full	$0.481 \pm 3\times10^{-3}$	$0.532 \pm 3\times10^{-3}$	$0.604 \pm 4\times10^{-3}$	10.6%	25.6%
Blocking	$0.00698 \pm 4\times10^{-5}$	$0.00696 \pm 4\times10^{-5}$	$0.00691 \pm 4\times10^{-4}$	0.3%	1.1%
High Risk	$1.15\times10^{-4} \pm 3\times10^{-7}$	$1.08\times10^{-4} \pm 3\times10^{-7}$	$9.55\times10^{-5} \pm 2\times10^{-7}$	5.9%	17.0%

Table 6.3: Steady-State Results and *Diff* Values, Bursts of 850

Variable	*Full Model value (m_1)*	*RV Passing value (m_2)*	*Avg. Passing value (m_3)*	*Diff* (m_1, m_2)	*Diff* (m_1, m_3)
Proc Time	$0.1064 \pm 7\times10^{-4}$	$0.1041 \pm 6\times10^{-4}$	$0.1053 \pm 7\times10^{-4}$	2.1%	1.1%
Arrivals	776 ± 3	786 ± 2	909 ± 2	1.7%	17.2%
Arrivals2	1629 ± 5	1737 ± 4	1983 ± 4	6.7%	21.8%
Full	$0.510 \pm 4\times10^{-3}$	$0.568 \pm 3\times10^{-3}$	$0.865 \pm 3\times10^{-3}$	11.4%	69.8%
Blocking	$0.00731 \pm 5\times10^{-5}$	$0.00721 \pm 4\times10^{-5}$	$0.00726 \pm 4\times10^{-4}$	1.4%	0.7%
High Risk	$1.18\times10^{-4} \pm 4\times10^{-7}$	$1.11\times10^{-4} \pm 3\times10^{-7}$	$9.82\times10^{-5} \pm 2\times10^{-7}$	6.1%	17.0%

Table 6.4: Steady-State Results and *Diff* Values, Bursts of 900

Variable	*Full Model value (m_1)*	*RV Passing value (m_2)*	*Avg. Passing value (m_3)*	*Diff* (m_1, m_2)	*Diff* (m_1, m_3)
Proc Time	$0.1103 \pm 7\times10^{-4}$	$0.1101 \pm 7\times10^{-4}$	$0.1106 \pm 7\times10^{-4}$	0.3%	0.2%
Arrivals	786 ± 3	794 ± 2	902 ± 2	1.1%	14.8%
Arrivals2	1653 ± 5	1773 ± 4	2001 ± 4	7.3%	21,1%
Full	$0.539 \pm 4\times10^{-3}$	$0.623 \pm 3\times10^{-3}$	$0.989 \pm 4\times10^{-3}$	11.4%	83.5%
Blocking	$0.00767 \pm 5\times10^{-5}$	$0.00766 \pm 4\times10^{-5}$	$0.00768 \pm 4\times10^{-4}$	0.2%	2.3%
High Risk	$1.22\times10^{-4} \pm 4\times10^{-7}$	$1.15\times10^{-4} \pm 3\times10^{-7}$	$1.02\times10^{-5} \pm 3\times10^{-7}$	5.4%	16.1%

Diff value (1.1%) for that same case of the Arrivals variables, and an error of 11.4% for the Full variable, which, although not great, is much better than 83.5%. However, both the Proc_Time and Blocking variables are dependent only on the number of arrivals being processed, and not on the frequency with which the system writes snapshots, making $T_\epsilon = \infty$ for these variables. Thus, it makes perfect sense that the average-passing model performs well on those variables, and we note that random-variable-passing also works well in those cases, as it should whenever the average-passing works well. For all the other variables, the number of snapshots written is important, and for that reason, those variables show a much better accuracy with random-variable-passing than with average-passing.

6.3 Duration Variable Results

The random-variable-passing model, with *Diff* values from 1 to 4% for the duration variables, performed much better than the average-passing model, with *Diff* values from 16 to 20%. The results are presented in Table 6.5, Table 6.6, and Table 6.7. Both of these measures depend on the number of arrivals since the last update, which is related to the time between updates. For the average-passing model, the time between updates is constant, but it varies for the random-variable-passing model. The constant arrival rate of the average-passing model misses the fact that a bursty interval will result in one or more updates (occurring before the timeout) with exactly the threshold value number of arrivals, while a less active interval will result in a smaller number of arrivals before the timeout forces an

Table 6.5: Duration Results and *Diff* Values, Bursts of 800

Variable	*Full Model value* (m_1)	*RV Passing value* (m_2)	*Avg. Passing value* (m_3)	*Diff* (m_1, m_2)	*Diff* (m_1, m_3)
Wait Time	$0.884 \pm 9\times10^{-3}$	$0.918 \pm 9\times10^{-3}$	$1.044 \pm 1\times10^{-3}$	3.0%	18.1%
Blocking	$0.06157 \pm 5\times10^{-4}$	$0.06348 \pm 5\times10^{-4}$	$0.0735 \pm 6\times10^{-4}$	3.1%	19.3%

Table 6.6: Duration Results and *Diff* Values, Bursts of 850

Variable	*Full Model value* (m_1)	*RV Passing value* (m_2)	*Avg. Passing value* (m_3)	*Diff* (m_1, m_2)	*Diff* (m_1, m_3)
Wait Time	$0.909 \pm 9\times10^{-3}$	$0.926 \pm 9\times10^{-3}$	$1.063 \pm 1\times10^{-3}$	1.0%	16.1%
Blocking	$0.06234 \pm 5\times10^{-4}$	$0.0646 \pm 5\times10^{-4}$	$0.0751 \pm 6\times10^{-4}$	3.6%	20.4%

update. Thus, more updates are modeled by the random-variable-passing model since some of the inter-update times will be less than timeout threshold, and there are fewer arrivals per update. The variation in arrival rate is critical in modeling the number of arrivals that are processed in an update and is not captured by the average-passing model.

Not only does the random-variable-passing model provide accurate values for the means of the duration variables, it also accurately measures the distribution of these variables, as seen in Figure 6.1. The random-variable-passing model gives a very tight fit to the full model distribution, while the average-passing model is much lower. If phase1, phase2, and phase4 are modeled as deterministic distributed activities, instead of exponentially distributed activities, the average-passing model does much worse, while random-variable-passing model does about as well as it did in the original case. Phase1, phase2, and phase4 could be reasonably modeled as deterministic distributed activities if the variability of request sizes was accounted for in the arrival process rather than the update process. The average-passing model distribution begins to look more like a step function for the adjusted model, while the random-variable-passing model can still capture the varying nature of the system. This

Table 6.7: Duration Results and *Diff* Values, Bursts of 900

Variable	*Full Model value* (m_1)	*RV Passing value* (m_2)	*Avg. Passing value* (m_3)	*Diff* (m_1, m_2)	*Diff* (m_1, m_3)
Wait Time	$0.911 \pm 9\times10^{-3}$	$0.949 \pm 9\times10^{-3}$	$1.078 \pm 1\times10^{-3}$	4.2%	18.4%
Blocking	$0.0633 \pm 5\times10^{-4}$	$0.0656 \pm 5\times10^{-4}$	$0.0751 \pm 6\times10^{-4}$	3.7%	18.7%

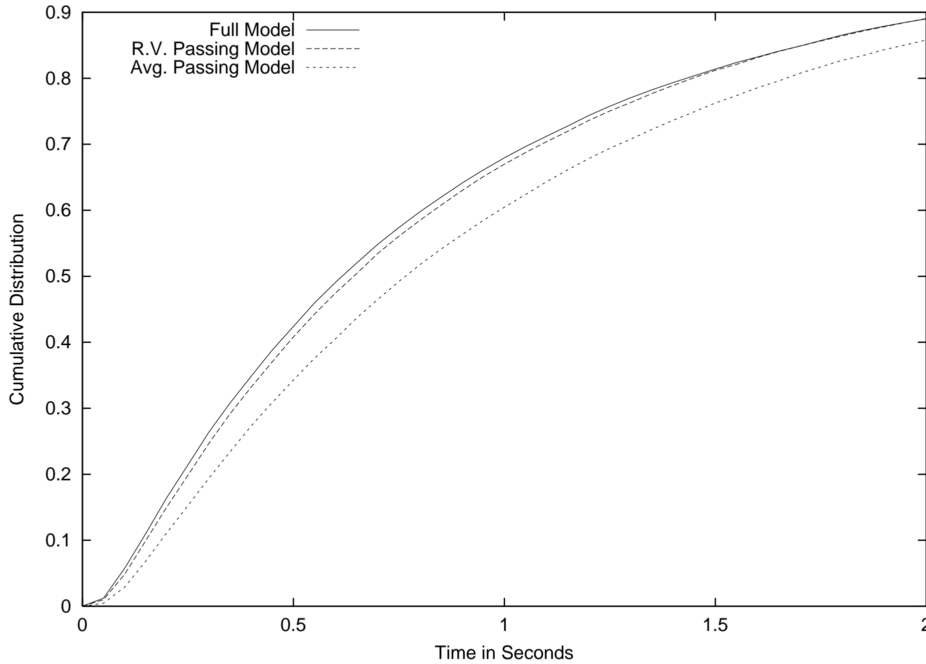

Figure 6.1: Distribution of Delays for Complete Snapshot for Bursts of 850

is clear even if only the most dominant action (phase3) is changed to have a deterministic distribution, as shown in Figure 6.2.

6.4 Summary of Results

Three models were solved to demonstrate two approximation techniques. While both of the approximate techniques sped up solution time of the models by about two orders of magnitude, they also added errors to the models. The average-passing technique is the one most likely to be used in a simple result-passing model. However, we found that while it is useful for a few of the measures we are interested in, it can lead to very high errors in others. The second approximation technique provided much better results, with errors of no more than 11%. That amount of error may seem large, and indeed it may be large depending on the accuracy requirements of the solution of the measures. However, the technique provides a general idea of how the process behaves and could be very useful. It not only has a lower error than the average-passing technique, but also represents more behaviors accurately. It

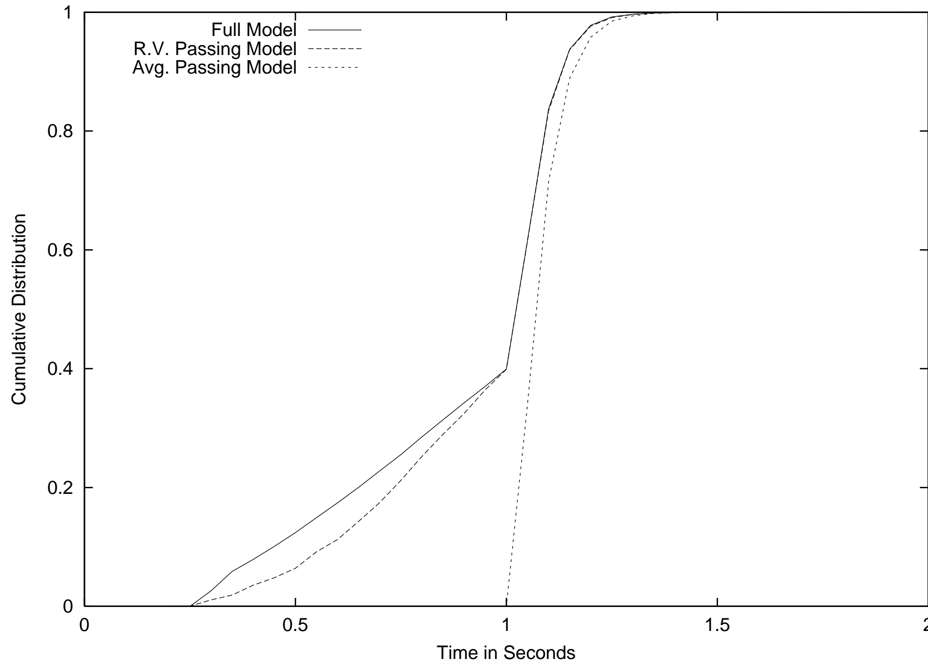

Figure 6.2: Distribution of Delays for Complete Snapshot for Bursts of 850, Deterministic phase3 Distribution

was able to measure the distribution of the measures accurately in a way that the average-passing technique could not.

These results do not mean that the random-variable-passing technique is useful for all models, or that the average-passing technique is good for none. It would be possible to construct a model that would have arbitrarily large error for the random-variable-passing technique, and another model that would get an almost exact answer for the average-passing technique. These results are evidence, however, of the limits of the average-passing technique and of the utility of the random-process-passing technique in situations where the average-passing technique does not work or is not known to work.

CHAPTER 7

CONCLUSIONS

This thesis presents connection as a decomposition technique that can be used to solve large and stiff models. After terminology was developed to describe models, some important submodels were introduced, as were different ways one model could approximate another model. Extending the existing connection infrastructure, a framework was developed and used to describe the use of connection with decomposition. A theory was developed to determine when connection could be used as a decomposition technique for a binary decomposition, and then extended to handle a decomposition into n submodels. This theory was then used to explain the behavior of a formalism (FiFiQueues) that uses result passing.

Next, the theory was applied to a special class of models. Models with isolated submodels were determined to be a class of models that could provide information on approximation techniques and results to pass that would be useful to our understanding of connection in general, not just connection applied to this class of model. Four schemes were developed and analyzed for the passing of results of varying levels of information. Two of these schemes, the random-variable-passing technique and the average-passing technique, were determined to be practical and provide a good speed-up/accuracy trade-off for certain models.

These two approximation techniques (random-variable-passing and average-passing) for models with isolated submodels were then evaluated via a case study of a real world file server, the NetApp line of file servers. It was determined that the process of file service requests formed an isolated submodel, allowing the application of the techniques. The solutions of the

models with these techniques show that the conditions for both techniques were met some of the time, but that the random-variable-passing technique was generally more applicable than the average-passing technique. Both provided speedups of about one to two orders of magnitude with varying levels of error. The random-variable-passing technique had an error of about 5% for most variables and no more than 11% for any variable, while the average-passing technique had errors ranging from a low of 1-2% for some measures to a high of up to 83% for other measures.

7.1 Future Work

A few areas of future research are created by this work. One such area is the development and analysis of approximation techniques for models with isolated submodels. Four techniques, two of which are known to be practical, have been developed for use with such models. However, they are not a canonical list of all possible approximations, and original work could be done on developing new approximations and evaluating their performance and usefulness. Further, a detailed analysis of the existing approximation techniques or new ones could be done for cyclic connected models. While the analysis of approximation techniques for models with isolated submodels provides an insight into the behavior of their behavior with cyclic connected models, it does not provide a complete understanding.

Another area for continued research is the classification of submodels. It was shown that an open queuing network with Poisson arrivals, two nodes, and exponential service time distribution was decomposable into error-minimizing submodels, and an idealized version of FiFiQueues was decomposable into passable submodels. There are many more types of models that we would wish to model using connection, and the ability to determine or prove which ones are decomposable into passable or error-minimizing models would be a useful contribution.

A third area of continued work is the automation of connection as a decomposition

technique. Currently, the use of connection as a decomposition technique requires that the modeler identify appropriate models, decompose them into the appropriate submodels, and add the needed measures. A theory on the automation of any of these steps would facilitate, and increase the use of connection.

REFERENCES

[1] M. Molloy, "On the integration of delay and throughput measures in distributed processing models," Ph.D. dissertation, University of California at Los Angeles, 1981.

[2] S. Natkin, "Reseaux de Petri stochastiques," Ph.D. dissertation, Conservatoire National des Arts et Metiers-PARIS, 1980.

[3] M. A. Marsan, G. Balbo, and G. Conte, "A class of generalized stochastic Petri nets for the performance analysis of multiprocessor systems," *ACM Trans. Comp. Sys.*, vol. 2, no. 2, pp. 93–122, May 1984.

[4] J. F. Meyer, A. Movaghar, and W. H. Sanders, "Stochastic activity networks: structure, behaviour and applications," in *Proc. International Workshop on Timed Petri Nets*, 1985, pp. 106–115.

[5] C. Lindemann, *Performance Modelling with Deterministic and Stochastic Petri Nets*, John Wiley and Sons, Chichester, England, 1998.

[6] C. M. Woodside, J. E. Neilson, D. C. Petriu, and S. Majumdar, "The stochastic rendezvous network model for performance of synchronous client-server-like distributed software," *IEEE Transactions on Computers*, vol. 44, no. 1, pp. 20–34, January 1995.

[7] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of Modeling and Simulation*, Academic Press, San Diego, second edition, 2000.

[8] W. H. Sanders, "Construction and solution of performability models based on stochastic activity networks," Ph.D. dissertation, University of Michigan, 1988.

[9] G. Chiola and G. Franceschinis, "A structural colour simplification in well-fromed coloured nets," in *Proceedings of the Fourth International Workshop on Petri Nets and Performance Models 1991. PNPM91.*, 1991, pp. 144–153.

[10] D. Obal, "Measure-adaptive state-space construction methods," Ph.D. dissertation, University of Arizona, 1998.

[11] D. D. Deavours and W. H. Sanders, "An efficient disk-based tool for solving very large Markov models," *Performance Evaluation*, vol. 33, pp. 67–84, 1998.

[12] D. D. Deavours and W. H. Sanders, ""on-the-fly" solution techniques for stochastic Petri nets and extensions," *IEEE Transactions on Software Engineering*, vol. 24, no. 10, pp. 889–902, October 1998.

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

[14] B. R. Haverkort and A. Ost, "Steady-state analysis of infinite stochastic Petri nets: Comparing the spectral expansion and the matrix-geometric method," in *International Workshop on Petri Nets and Performance Models*, 1997, pp. 36–45.

[15] B. Tuffin and K. S. Trivedi, "Implementation of importance splitting techniques in stochastic Petri net package," in *Computer Performance Evaluation: Modelling Techniques and Tools: Proceedings of the 11th International Conference, TOOLS 2000, Schaumburg, IL*, 2000, pp. 216–229.

[16] H. H. Ammar and S. M. R. Islam, "Time scale decomposition of a class of generalized stochastic Petri net models," *IEEE Transactions on Software Engineering*, vol. 15, no. 6, pp. 809–820, June 1989.

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

[18] F. Gabbay and A. Mendelson, "The "smart" simulation environment - a tool-set to develop new cache coherency protocols," *Journal of Systems Architecture*, vol. 45, no. 8, pp. 619–632, February 1999.

[19] H. Stone and D. Thiebaut, "Footprints in the cache," *ACM Transactions on Computer Systems*, vol. 5, pp. 305–329, November 1986.

[20] A. Christensen, "Result specification and model connection in the Möbius modeling framework," M.S. thesis, University of Illinois, 2000.

[21] W. H. Sanders, "Integrated frameworks for multi-level and multi-formalism modeling," in *Proceedings of PNPM'99: 8th International Workshop on Petri Nets and Performance Modeling, Zaragoza, Spain*, September 8-10 1999, pp. 2–9.

[22] D. Daly, D. D. Davours, J. M. Doyle, P. G. Webster, and W. H. Sanders, "Möbius: An extensible tool for performance and dependability modeling," in *Computer Performance Evaluation: Modelling Techniques and Tools: Proceedings of the 11th International Conference, TOOLS 2000, Schaumburg, IL*, 2000, pp. 332–336.

[23] D. D. Deavours and W. H. Sanders, "Möbius: Framework and atomic models," in *Proceedings of the 9th International Workshop on Petri Nets and Performance*, September 2001, to appear.

[24] G. Ciardo and R. Zijal, "Well-defined stochastic Petri nets," in *Proc. 4th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 1996, pp. 278–284.

[25] R. Sadre and B. R. Haverkort, "FiFiQueues: Fixed-point analysis of queueing networks with finite-buffer stations," in *Computer Performance Evaluation: Modelling Techniques and Tools: Proceedings of the 11th International Conference, TOOLS 2000, Schaumburg, IL*, 2000, pp. 324–327.

[26] R. Sadre, B. R. Haverkort, and A. Ost, "An efficient and accurate decomposition method for open finite- and infinite-buffer queuing networks," Tech. Rep., Rheinisch-Westfälische Technische Hochschule, 1999.

[27] R. Sadre and B. Haverkort, "Characterising traffic streams in networks of MAP|MAP|1 queues," in *Proceeding 11th GI/ITG Conference on Measuring, Modelling and Evaluation of Computer Communication Systems, VDE Verlag*, B. Haverkort, Ed., 2001, to appear.

[28] G. Clark, T. Courtney, D. Daly, D. D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster, "The Möbius modeling tool," in *Proceedings of the 9th International Workshop on Petri Nets and Performance*, September 2001, to appear.

[29] A. Williamson, "Discrete event simulation in the Möbius modeling framework," M.S. thesis, University of Illinois, 1998.

[30] A. Watson and P. Benn, "Multiprotocol data access: NFS, CIFS, and HTTP," Tech. Rep. TR3014, Network Appliance, 1999.

[31] S. R. Kleiman, S. Schoenthal, A. Rowe, S. H. Rodrigues, and A. Benjamin, "Using NUMA interconnects to implement highly available filer server appliances," Tech. Rep. XP1004, Network Appliance.

[32] D. Hitz, J. Lau, and M. Malcolm, "File system design for an NFS file server appliance," Tech. Rep. TR3002, Network Appliance, 1995.