STATE-SPACE GENERATION TECHNIQUES IN THE MÖBIUS MODELING FRAMEWORK

BY

JOHN MARK SOWDER

B.S., University of Illinois, 1997

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, 1998

Urbana, Illinois

ABSTRACT

In modeling, analytical solutions to models of complex systems can often be obtained by creating the state-space of a model and then solving this behavioral representation using numerical techniques. However, in generating such state spaces, two difficulties arise: the prohibitive amount of computer memory needed to store large state spaces, and the large amount of time needed to generate the state-space. In addition, the algorithms used for state-space generation are generally tailored only to a specific modeling formalism. The work presented here addresses these issues using a new toolkit named Möbius. The goal of the Möbius project is to develop an object-oriented, formalism-independent, stochastic modeling framework, and implement the framework in a practical, usable performance/dependability evaluation tool.

Use of the Möbius framework permits the state-space generator to be implemented in a formalism-independent way, since the framework defines the properties of state and actions. Exponential, deterministic, and instantaneous action types are supported in the implementation of the state-space generator. To reduce the memory required and the time needed for state generation, efficient data structures and fast output file generation methods were used. The performance of the state-space generator implementation is promising and allows the analysis of large and complex models. By using the Möbius framework, the state-space generator implementation also provides a basis for easy extensions and feature addition.

To my wife, for her patience,
love, and understanding

ACKNOWLEDGMENTS

During my time at the University of Illinois, I have enjoyed the friendship and guidance of many special individuals. First, I would like to thank my advisor, Dr. William H. Sanders, for his technical guidance, his patience, and all his help with the Möbius project. I would also like to thank Dr. Sanders for letting me be a part of the Performability Engineering Research Group. Without that opportunity, I would not have gone to graduate school. I would also like to thank all my fellow workers in room 229 of CSRL. Thanks to Jay Doyle, Alex Williamson, and Gerard Kavanaugh for all your help, friendship, and technical guidance. Thanks also to Doug Obal and Dan Deavours for their invaluable help in answering my many technical questions. Special thanks to Jenny Applequist for her help in reviewing this thesis.

Additional thanks go to the Defense Advanced Research Projects Agency Information Technology Office, for funding under contract DABT63-96-C-0069, and Motorola Space Systems Technology Group, including Renee Langefels and Peter Alejandro, for their long-term funding of the *UltraSAN* and Möbius projects.

I also want to extend a very sincere thank you to my parents for always listening, supporting, and caring. Finally, I want to thank my wife Denise for her continuous encouragement, love, and understanding.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

1. INTRODUCTION

1.1. Modeling Overview

The use of prototypes is common in industry for the analysis of complex systems. The benefit of building a prototype is that it can effectively mimic the behavior of the system before final production. This allows for the analysis of the performance and dependability of the system before commitment to a final design. However, modern systems are often too complex to prototype in the early development stages. For these systems, computer modeling and analysis are also useful. By using the solutions to a model of a system, a designer can predict performance and diagnose problems and flaws earlier than would be possible if prototyping were the sole evaluation method.

Generally speaking, there are three modeling solution methods used to compute information about a system being modeled: closed-form solutions, numerical solutions, and simulation. Closed-form solutions are mathematical descriptions of the measured property written as a function in closed form. While using a closed form is ideal, it is generally impossible to determine one for all but the simplest models. Conversely, when a closed-form solution is not available, a simulation may be used to examine the model. Simulation is a way of numerically exercising the model for the inputs in question to see how they affect the output measures of performance [1]. In other words, a simulation samples the random process and observes the resulting behavior. With enough samples, the measures can be estimated with a certain level of confidence. While simulation can solve any model through this "brute force" approach, it becomes less practical for estimating measures based on events that are rare, because of the large number of samples necessary.

An intermediate approach to model analysis is to use a stochastic process to represent the system and solve the process numerically. These processes are often expressed with a high-level modeling formalism such as stochastic activity networks (SAN) [2] and converted into a state-level representation through a method called "state-space generation." *State-space generation* is an algorithmic way of producing the desired stochastic process of a model given

some higher-level, more abstract representation of the model. This behavioral model representation is then solved by one of several numerical techniques depending on characteristics of the behavioral model and on the type of measure the modeler is interested in analyzing. This solution method is called a *numerical solution method* and is important in model analysis because it produces fast, extremely accurate results for moderately complex systems with rare events, for which simulation is impractical. In addition, with moderately complex systems, numerical results can often be produced with higher accuracy in less time than they could be with simulation.

1.2. Möbius Overview

Many modeling tools are currently available that can be used to represent a system in a compact high-level representation, generate a state-space from the high-level representation, and then use numerical solutions or simulation to determine properties of interest. Examples of such tools include *UltraSAN* [3], *HiQPN* [4], and *DSPNexpress* [5], among others. Each of these tools is based on a single high-level modeling language or *formalism.* The algorithms used for solution and analysis of these modeling languages may be fast, but are often tied closely to the high-level representation that the particular tool uses. Ideally, a modeling tool would have the ability to use many modeling languages to represent systems and use a generic yet complete set of modeling primitives. The SHARPE modeling tool [6] allows multi-formalism models, but limits interactions between models to results passed after model solution, and does not permit sharing of states and events between models. Result sharing allows solution methods specific to each formalism to operate on each model, but does not allow general solution methods to operate on a combined, multi-formalism model. The Möbius modeling framework allows more general interaction between models, while at the same time supporting multiple formalisms and a wide variety of solvers.

The Möbius approach does not prescribe a particular modeling formalism, but rather a framework to support a diverse set of formalisms. As long as a formalism can be expressed in the framework, it can be supported in Möbius. The Möbius framework consists of a basic set of modeling primitives including “models,” “states,” and “actions.” Each basic component has associated functions defined on it to allow for model execution. Using the framework,

each component can be used to define how a model using a specific formalism can act. Thus, the Möbius framework says what the components can do, not how they do it. For example, "firing" an action in a model changes state, but how it changes state depends on the formalism, the model, and how the "firing" was defined.

The Möbius framework components have been implemented in a tool referred to as the *Möbius tool*. The Möbius tool is written in C++ and Java. Java was chosen for its property of rapid interface development due to its assortment of predefined classes and libraries. C++ was chosen for the state-space generator since C++ is compiled and thus produces code that executes fast.

1.3. Research Objectives

Four areas of research within the Möbius tool are presented in this thesis: compact and efficient data structures for state generation that produce fast execution and low memory usage; a formalism-generic state generation algorithm, support for deterministically and exponentially distributed and instantaneous actions, and an interface in the Möbius tool that aids in producing the state space of a high-level model. The main objectives of the work are to examine performance results of the new algorithms as well as to provide theory and insight into additional state generation techniques.

The rest of the thesis is organized into five chapters. The second chapter presents a more detailed explanation of the Möbius modeling framework, highlighting its interaction with the state-space generator. The basic modeling building blocks are explained there, as are all of the model types within the Möbius tool. The second chapter also includes an overview of stochastic processes for exponential and deterministic distributions. The third chapter describes and explains the formalism-generic state generation algorithm. In Chapter 4, a detailed description of the implementation of the state generation application is presented. Chapter 4 emphasizes the data structures used in the implementation. In Chapters 5 and 6, the results, conclusions, and future work are discussed. In Appendix A, the user's manual for the state-space generator tool is presented. Finally, Appendix B details the output file formats that can be produced by the Möbius state-space generator.

2. MÖBIUS FRAMEWORK

The Möbius framework defines an extensible modeling environment that supports a diverse set of formalisms. A fundamental part of the Möbius modeling framework is the "formalism-generic" interaction between the model and the solution mechanisms. Of specific interest to the state-space generator is the path from "atomic" model to "solvable" model. In this chapter, definitions and explanations of each model type in the Möbius framework are presented. Also, the basic model components "state" and "action" are explained. Finally, in the last section of this chapter, the output of the state-space generator, a stochastic process, will be reviewed to help the reader understand the state generation algorithm described in the next chapter.

2.1. State Variables

State variables are entities that have a value and a type associated with each of them and are the basic building blocks of a model. An example of a state variable is a place in a stochastic activity network, where the value is the marking of the place and the type is a natural number. In a model, the set of all state variables and their values represents the "state" of the system. In the Möbius tool, for example, functions used on state variables include `statesize()`, `setstate()`, and `currentstate()`. These methods define the physical size the state occupies in the executable version of the model (memory size), a method to set the state variable to a particular value, and a method to return the value of all the state variables in a model at a given time, respectively.

A key feature of this definition of state in the Möbius framework and tool is that it allows for state to be implemented in a potentially complex way. For example, the state of a model could be defined as the collection of all markings in a stochastic activity network combined with the values in a queuing network. The only restriction is that the value and the type of the state variables must be precisely defined.

2.2. Actions

Actions are the state-changing entities of a model. Within the framework, support is provided both for *timed actions*, which are actions that complete in a specified amount of time, and for *instantaneous actions*, which are actions that complete in zero time. The exact implementation of the action specification, including details of the enabling functions, distribution parameters, and state-changing functions, is left to specific formalisms. The Möbius tool simply provides a set of data members and methods that must be defined on the model primitives by the formalism designer in order to describe the execution of models within the formalism. For example, in a stochastic activity network, the enabling function, reactivation methods, and distribution type must be defined in order for a SAN model to execute.

An important aspect of actions in the Möbius environment used by the state-space generator is the use of "ranks" and "weights." The *rank* of an action is defined as the priority of an action within a "group." The *weight* of an action is the relative importance of an action as compared to other actions in a "group." In state generation, weights are used to probabilistically determine certain exploration paths. For example, in a SAN, weights are the equivalent of cases on activities. Ranks are used by the state-space generator to break ties when two or more instantaneous actions are "enabled" in the same state of the system. An action is said to be *enabled* if in the current state the action is allowed to fire or execute the state-changing function associated with it through the Möbius framework. Chapter 3 explains in more detail the use of ranks and weights within the state generation algorithm.

A useful extension available as a result of the existence of the rank and weight attributes of actions is the creation of action "groups." *Groups* are sets of actions combined together for selection purposes. Groups are mainly used in the context of simulation of the model, as described in [7]. In simulation, groups behave as actions, but selection of specific actions within the group may occur as a result of model execution. Action composition allows for policies to be developed that define when actions within groups are selected and the processes by which the selection occurs. In the context of the state-space generator, groups are used in the calculation of the total weight of a group when determining the probability of an action firing, as explained in Chapter 3. For example, in stochastic activity networks,

activities have cases, which are used to choose probabilistically between possible paths of execution. In Möbius, a SAN activity with cases is represented as a group of activities such that each activity's weight corresponds to the case probability.

2.3. Models

Within the Möbius framework, models are made up of actions and state variables. Four types of models are supported in the Möbius framework: "atomic," "composed," "reward," and "connected." *Atomic models* are the basic building blocks of any larger models that are developed using only one formalism. Combinations of models using single or multiple formalisms are called *composed models*. Composed models allow creation of new complex system models, using the most appropriate formalism for each component. Not only do composed models facilitate the specification of complex combinations of models, but they also enable exploration of the model structure to obtain a more time- and/or space-efficient solution. Models joined through results are called *connected models*. Connected models can be used for model interaction through results passed after model solution, as found in SHARPE [6].

Defining "reward variables" on these composed or atomic models allows the various analytic and simulation solution methods to solve for the behavioral characteristics of the model. *Reward variables* are a method for defining functions on the state of a model. A model containing reward variables and other model(s) is called a *reward model*. Reward models contain methods for specifying reward variables in terms of the underlying model's state. Reward models may also have additional notions of state. For example, using stochastic activity networks, a reward model may add state to the model for storing impulse reward information.

Once reward variables are defined on a model, the model is said to be *solvable.* Certain parameters within a solvable model may be varied during execution by creation of *studies*. These studies allow variables within the model to change; each variation is called an *experiment*. Thus each study may contain multiple experiments that change the model's parameters such that the user can analyze the model under many different conditions. Analytical solution methods and simulations may then be used to analyze the derived solvable

model and the experiments defined on it. The state-space generator uses a solvable model to generate a stochastic process, which can be solved numerically, as is explained in detail in the next section.

2.4. Stochastic Process Overview

Through the state-space generation, a stochastic process representing the behavior of a solvable model is created. Then, well-established numerical techniques can be used to solve this stochastic process for measures of interest as specified by the reward variables of the model. While many types of stochastic process exist, only a few can be solved numerically. This section will describe two stochastic process types that can be generated by the Möbius state-space generator: Markov processes and Markov regenerative processes. A complete formal definition of the Markov and Markov regenerative processes will not be presented. Instead, a brief overview of the mathematical background will be provided, and details needed for state generation will be examined.

Stochastic processes are mathematical models useful for the description of random phenomena as functions of a parameter that usually has the meaning of time (exclusively time for the Möbius tool). More specifically, a stochastic process is a family of random variables $\{X(t), t \in T\}$ defined over the same probability space, indexed by the parameter t, and taking values in the set S. The values assumed by the stochastic process are called *states*, so that the set S is called the *state space* of the process [8].

One type of stochastic process generated by the Möbius state-space generator represents a Markov process. Markov processes have the following property:

$$P\{X(t) \le x \mid X(t_n) = x_n, X(t_{n-1}) = x_{n-1}, ..., X(t_0) = x_0\}$$
$$= P\{X(t) \le x \mid X(t_n) = x_n\} \quad t > t_n > t_{n-1} > ... > t_0$$

which is known as the *Markov memoryless property*. A stochastic process for which this condition holds is known as a Markov process. In words, the condition says that the future evolution of the process from the instant of time t_n on is independent of the past history of the process; the future depends only on the current state of the process. Models containing only timed exponentially distributed and instantaneous actions produce Markov processes.

Another important class of processes produced by the Möbius state-space generator is that of Markov regenerative processes. A Markov regenerative process has the property that at some time points the process (probabilistically) restarts itself [9]. Informally, there is a sequence of embedded time points at which the Markov memoryless property is satisfied. At these time points, future behavior of the process is independent of the past behavior. Models containing instantaneous actions and timed actions with exponential and deterministic distributions produce Markov regenerative processes. The only limitation imposed on a model is that only one deterministic distribution can be enabled for any state of the model.

When numerical techniques are used to solve a model, specific details of the Markov or Markov regenerative process are needed. For the Markov process representation, the state-space generator partially outputs the state transition-rate matrix (sometimes called the infinitesimal generator matrix) Q, defined as

$$Q = [\, q_{ij} \,] \qquad i, j \in S$$
$$q_{ii} = -\sum_{j \neq i} q_{ij} = -q_i$$

where q_{ij} is the rate of the process from state i to state j. The state-space generator does not directly produce the q_{ii} entries since they are easily obtained by a summation of the q_{ij} entries. This form allows the state occupancy probability vector (a vector containing a list of all the probabilities of being in a certain state of the process at a given time) to be solved using a variety of numerical techniques for both steady-state and transient solutions.

The output representation produced by the state-space generator for a Markov regenerative process is similar to that produced for a Markov process. This representation is solver-specific and was originally implemented in [10]. To express the regeneration points, a similar state transition-rate output is produced in which the probabilities of each next tangible state are output. Thus the output is a hybrid state-transition rate matrix with rates being replaced by probabilities when a deterministic action is enabled. These probabilities are appended with a minus sign so that the numerical solver can distinguish rates from probabilities.

3. STATE GENERATION ALGORITHM

In this chapter, a detailed description of the state generation algorithm used to generate the underlying stochastic process from a model will be explained. A formalism-generic algorithm is presented in terms of the methods defined in the Möbius framework. In the context of this chapter, the term *method* is used as a way of compactly expressing details of the algorithm, not implementation-specific functions. The state generation algorithm is divided into two parts. The first part is the main state generation algorithm that determines all states of a model in a breadth-first manner. The second part specifies precisely how to determine next states when timed and/or instantaneous actions are enabled in a specific state.

```
1     A_t = {α_t0, α_t1, ...}      // A_t is the set of all timed actions in the model
      A_z = {α_z0, α_z1, ...}      // A_z is the set of all instantaneous actions in the model
      S = {∅}                      // S is the set of generated states
      U = {μ_0}                    // U is the set of unexplored states, μ_0 is the initial state
      NS = {∅}                     // NS is the set of next states
5     While U ≠ ∅
6
            For some μ ∈ U
                  NS = {∅}
                  NS = ComputeNS(μ , p=1)  using Algorithm 2
                                    // p is a probability
                  ComputeReward(μ)
10                ∀ ns ∈ NS
                     if ns ∉ S
                        S = S ∪ ns
                        U = U ∪ ns
                  End ∀
15                U = U - μ
            End For

      While end
```

Figure 1: State-Space Generation, Algorithm 1

Algorithm 1 (see Figure 1) uses five sets to determine all states in a model: the set of timed actions (A_t), the set of instantaneous actions (A_z), the set of "generated" states (S), the set of "unexplored" states (U), and the set of next states (NS). The sets of timed and instantaneous actions are used in Algorithm 2. The explanation of these two sets is left until the explanation of Algorithm 2. The set of "generated" states contains all unique states that have been produced at any point in time by the state-space generator. *Generated* states are those "tangible" states that have been found using the *ComputeNS* method. A *tangible* state is a state that has no instantaneous actions enabled in it. The *unexplored* set contains all tangible states that have not been used by Algorithm 2 to determine all tangible next states. The set of next tangible states found by using Algorithm 2 is stored in the *NS* set. These five sets are used to determine all tangible states for a model.

After initialization of the five sets in the algorithm, the main while loop in which unexplored states are generated is entered. First, some state in the unexplored set is picked. Initially, this is the initial state of the model. Then, Algorithm 2 is used to determine all tangible next states of a state μ through the method *ComputeNS*, which takes as input a state and a probability. Initially, the probability value is set to 1 to determine rates from one tangible state to another as explained in Algorithm 2. Algorithm 2 is presented in Figure 2.

Before explanation of the policies for execution of Algorithm 2, the *Probability* method will be presented. The *Probability* method is used to determine the probability that an action will fire in a certain state of the model. *Probability* takes an action as input and returns the probability of this action firing in the current state of the model. The probability of each enabled action is determined using the weight method on an action and the equation below.

$$Probability(\alpha_z) = \frac{\alpha_z.Weight()}{\alpha_0.Weight() + \alpha_1.Weight() + \alpha_2.Weight() + ... + \alpha_z.Weight()}$$

The probability is used to determine rate values between tangible states. The *StoreQ* method is called to store the rates in the state-transition rate matrix representation output by the state-space generator. When the model produces a Markov process (i.e., only exponential timed and instantaneous actions are present) then the rate to a next state can be represented by the multiplication of all probabilities from one tangible state to another by the rate of the

action enabled in the previous tangible marking. In the current implementation, two or more

```
1     ∀ αt ∈ At
        If αt.Enabled()
          μi+1 = αt.Fire(μi)
        p = p*Probability(μi)
        If Az ≠ ∅ and PathLength() < MaxPathLength
5       ∀ αz : αz.Enabled()
          While αz.NumberEnabled() > 0
            Case 1:                    // one enabled
                μi+1 = αz.Fire(μi)
            Case 2:                    // more than one enabled
10              ∀ αz ∈ Az : αz.Enabled() : αz..Rank() = αz+1.Rank()
                  SetState(μi)
                  μi+1 = αz.Fire(μi)
                  goto line 4, μi =μi+1, p = p*Probability(αz)
                End ∀
15              Else
                  αz = HighestRank(αz, αz+1)
                  μi+1 = αz.Fire(μi)

          End While
        End ∀
        Else StoreQ(p)
20      End ∀
```

Figure 2: Next Tangible State Computation, Algorithm 2

instantaneous actions enabled by a state change from a deterministic action are not allowed.

Algorithm 2 shows how the set of all next states is computed given a state of the model. The algorithm uses the sets A_t and A_z. These sets contain all the timed and instantaneous actions of a model. These actions are split into two sets, since different execution methods are needed for each type. If the current state of the model only contains enabled timed actions, then the next state is determined by calling the *Fire* method on an action. The *Fire* method on an action takes a state as input and returns the new state by executing the action's state changing function. The method is executed for every timed action

that is enabled given the state μ of the model. However, if instantaneous actions are enabled, a more complex state-changing algorithm is employed.

While instantaneous actions are enabled, the model continues to change state according to the specific instantaneous action being fired. With respect to the firing of instantaneous actions, there are two possible cases: there is one enabled instantaneous action, or there are two or more enabled instantaneous actions. If two or more instantaneous actions are both enabled in the current state, then an "execution policy" must be used to determine which one is allowed to fire. The Möbius framework defines the *execution policy* in the context of the state-space generator as the process in which model execution is determined when instantaneous actions are enabled. The Möbius state-space generator uses ranks and weights in this policy. Specific to the algorithm, if more than one instantaneous action is enabled, the first step in the model execution is that the method *Rank* is called on the action. The *Rank* method returns the rank of an instantaneous action within the model. The instantaneous action with the highest rank is then fired. However, if two instantaneous actions have equivalent ranks, then Algorithm 2 is recursively applied until the next tangible state is determined. The variable *p* is modified at each recursive step by multiplication of the new probability found through the *Probability* method by the old value. This value is then used in determining the overall rate for a Markov process, or probabilities for a Markov regenerative process, from one tangible state to another.

When only one instantaneous action is enabled, the *Fire* method executes and returns the new state. Once the new state is returned from this part of the algorithm, the check for any enabled instantaneous actions, line 4, is repeated. The algorithm then recursively continues while any new instantaneous actions are enabled until all tangible next states are found. The number of recursions, however, is limited to *MaxPathLength* to allow for practical implementation. After all tangible states are found, Algorithm 2 returns the set of all next states of the model given a current state as input.

After returning from Algorithm 2 with the set of next states, Algorithm 1 first calls the method *ComputeReward* with the current state μ as input to determine the reward of state μ of the model. The *ComputeReward* method is defined on a reward model in the Möbius framework, which allows for formalism-independent reward gathering. After reward

computation, all next states determined by Algorithm 2 not yet explored are added to the unexplored set. This algorithm continues until the unexplored set is empty and thus all tangible states of the model have been determined. The algorithm returns the set *S* which contains the set of all tangible states of the model. The implementation of the state-space generator algorithm in the Möbius framework is presented in Chapter 4.

4. STATE-SPACE GENERATOR IMPLEMENTATION

In this chapter, we will discuss the implementation of the state-space generation algorithm presented in Chapter 3. The Möbius state-space generator implementation has two main components: a C++ solution engine and a Java interface. As shown in Figure 3, the execution of the C++ state-space generator solution engine is done in three phases: initialization, state and reward generation, and output formatting. We first discuss the initialization stage, which performs the setup required for the model interaction and initializes the data structures. The next section of the chapter discusses the state and reward generation execution stage. In that section, a more detailed description of the data structures used in the implementation as well as discussion of the implementation of deterministic actions is presented. The output-formatting stage is then discussed. That section describes the different representations for the state-transition rate matrix produced by the state-space generator. The chapter concludes by discussing the Java interface used to interact with the C++ solution engine.

Figure 3: Execution Flow Diagram

4.1. Initialization

In the initialization stage, several steps occur to set up values and parameters needed by the state-space generation routine. The initialization steps in the state-space generator implementation are important since they set up the generic interaction between a model and the state-space generator, and more specifically the interaction with actions in the model. As shown in the first algorithm of Chapter 3, the state-space generator needs two lists

corresponding to the two kinds of actions: timed and instantaneous. These lists are formed by using data members defined on actions that flag the type of action being represented. Once the lists are made, the list of timed actions is scanned to determine the distribution types of the actions. Determination of a model's action types allows the state-space generator to ascertain whether all actions are of types supported in the state-space generator. Determination of the action types also makes it possible to use optimized routines for models with certain actions.

The initialization stage also allocates an initial block of memory to each data structure. The size of these blocks is fairly large, to avoid the significant overhead that would be caused by many additional allocations during state-space generation. The state-space generator also initializes the output files used to store the state-transition rate matrix and associated reward. In order to minimize the amount of data that must be kept in main memory, these files are written to disk during the state and reward generation stage of state generation. The files are then converted to the desired output format in the output-formatting stage.

4.2. State and Reward Generation

After initialization, the next stage of execution is the state and reward generation. Three issues are discussed in this section with respect to the state and reward generation stage: the interaction between models and the state-space generator, the data structures used to store and look up states, and the issues related to supporting deterministically distributed actions. More specifically, the first subsection describes the generic interface of the state-space generator that allows the formalism-independent state generation algorithm to be implemented. The second subsection presents the two data structures used in the generation algorithm: a hash table and a queue. Finally, we discuss how deterministic actions are supported in the state-space generator.

4.2.1. Model interaction

The Möbius state-space generator interacts with models in a formalism-independent way. Three methods are used to do this: `SetState()`, `CompareState()`, and `Fire()`. `SetState()` sets the current state of the model to the input passed to this method.

`CompareState()` takes two states as input and returns true or false depending on whether they are equal. `Fire()` takes an action as input and returns the new state of the model after this action is fired. The state-space generator obtains implementations of these methods from the reward model. These methods, defined on the reward model, allow the state-space generator to interface indirectly with many types of models, including composed and atomic models.

In addition, the state-space generator interacts with the reward model by use of two methods, `Reward()` and `Impulse()`, that compute the reward associated with particular states in the model. The two methods are the implementation of *ComputeReward* as used in Algorithm 1 of Chapter 3. Specifically, the `Reward()` method returns the reward rate for a specific state. The `Impulse()` method returns the impulse reward associated with the firing of a certain action in a model. During the state-space generation, the values returned by the `Impulse()`and `Reward()` methods are output to a file to avoid the need to hold the set of rewards for all states in memory at once. After the state and reward generation stage has completed, the file contains all reward values for every reward variable for every state in the model.

4.2.2. Data structures

A main limitation of state-space generation of large models is the prohibitive amount of time and memory needed. The choice of particular data structures to use in the state generator is thus very important. This section explains the data structures used to store the set of generated and unexplored states. Due to its fast lookup and insertion time, a hash table is used to store the set of generated states due to its fast lookup and insertion time. A queue is used to store the set of all unexplored states, since it provides constant insertion and deletion time.

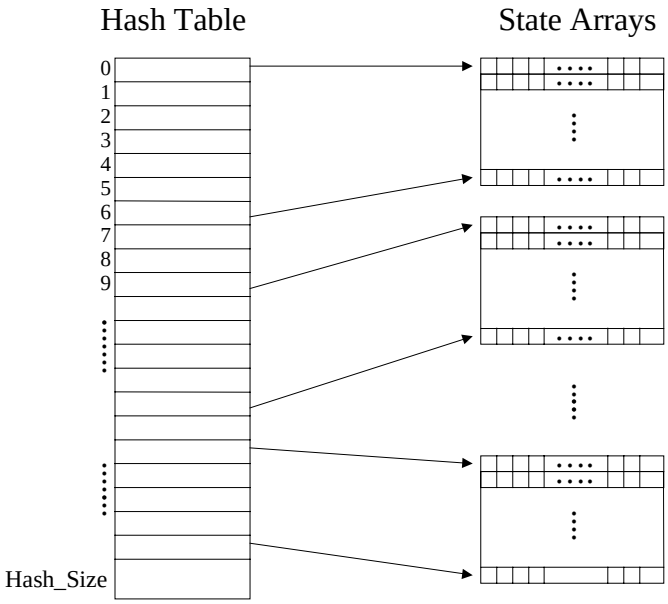

Figure 4: Hash Table Data Structure

Hash Table

The data structure used to store the set of generated states must have fast lookup and fast insertion time. Fast lookup time is needed since a lookup must occur for every state generated to determine whether the state is in the set of generated states. Also, each new state generated must be inserted in the generated set to be used in future comparisons. In addition to being fast, the data structure used to store the set of generated states must use as little extra memory as possible, beyond that required to store the states themselves. A hash table was chosen for this data structure because given an efficient hash function, it has fast lookup, fast insertion, and uses memory conservatively.

A pictorial representation of the hash table and associated data structures is shown in Figure 4. The figure shows the two components of the data structure: the hash table and the state arrays that contain the states themselves. The hash table holds 32-bit values used to determine where a state can be found in the state array structures. The state arrays are of fixed size and contain the states. Each state array has an associated ID. By using this ID and an index into the array, it is possible to locate particular states. Specifically, the 32-bit values

stored in the hash table contain the ID and index information needed. The first 24 bits (bit positions 0-23) of each value represent the index into a state array, and the remaining 8 bits (bit positions 24-31) represent the state array ID.

To determine the *hash key*, which is the index into the hash table to use when either storing newly generated states or searching for a specific state, we use a hash function on the state. To preserve the time efficiency of the hash table data structure, it is important to find a hash function that produces few collisions. Collisions occur when use of the hash function on two states produces two identical outputs, and thus two identical indices into the hash table.

It is not straightforward to determine a hash function for the Möbius state-space generator that produces few collisions, since no information about the structure of the state is accessible to the state-space generator and the size of the state being hashed is not known in advance. To build an efficient and useful hash function given these constraints, one must define a function that considers the state as a block of memory, but assigns a known structure to the memory. This is done by breaking each state into a set of 1-byte values. These values are then used as input to the hash function. The hash function also depends on the order of the 1-byte blocks and is adaptable to any state size.

In particular, the hash function computes a polynomial function (of degree 32) by use of Horner's rule [11]. The hash function is defined as

$$HashKey = \sum_{i=0}^{StateSize-1} State[StateSize - i] \bullet 32^{i}$$

where *StateSize* is the number of 1-byte blocks of state and *State* contains the 1-byte state values. The hash function computes a 32-bit key by summing all values from 0 to *StateSize-1*. At each step of the summation, the 1-byte state values are multiplied by 32 raised to the power of *i* to distribute the 1-byte state values over the 32 bits of the hash key. After the key is computed, the index into the hash table is taken to be the modulus of the hash key and the hash table size. The modulus is taken to produce an index within the size of the hash table. It should also be noted that the hash table size is a prime number so that the hash function will distribute well over the entire hash table [11]. In the actual implementation of the state-space generator, the summation is replaced with a bitwise exclusive-or to reduce the time necessary

to compute the key. If the state size is not larger than 6 bytes, then the hash function is directly executed, varying *i* from 0 to the *StateSize-1*, to obtain the hash key.

However, if the state has more than six 1-byte values, then some information may be lost due to the multiplication. Thus, in the Möbius state generator, each state must be brought within the range, which is 6 bytes. For this purpose, the state is broken up into 6-byte partitions, upon each of which the hash function is then executed. The hash key is then formed by taking the exclusive-or of the output of the hash function for every one of these 6-byte partitions.

Even though a relatively collision-free hash function is implemented in the state-space generator, a collision policy must be used when states hash to the same position in the hash table. To solve this problem, the state-space generator implements a quadratic probing collision resolution strategy using the function

$$f(j) = f(j-1) + 2j - 1$$

where j is the number of collisions. This function specifies a new index to go to when a collision occurs. For insertion, new indices are computed until an empty position is found. For a lookup, all states that are reached must be compared to the next state until either a match occurs (the state is not a new state), or an empty position is reached (the state is a new state, and should be added to the table).

If a hash table size that is a prime number is used, the quadratic probing procedure described in the last paragraph is guaranteed to terminate if the table is at least half empty [11]. If a prime number hash-table size is not used, the procedure may loop and never terminate. For this reason, rehashing (described next) is done when the table becomes half full. Quadratic probing also has the desired property of reducing "primary clustering." *Primary clustering* [11] occurs when keys hash into a cluster of values in the hash table requiring several attempts to resolve the collision, thus reducing overall performance.

Another important design decision when using a hash table is the hash table size. Unfortunately, with state generation, there is generally no way to determine *a priori* the number of states that will be generated before the states are actually produced. Thus, the state-space generator needs to be able to reassign hash table sizes dynamically. This dynamic

resizing is called *rehashing*. When the hash table fills to a specified level (usually 50% or less; the percentage can be specified as an input parameter to the generator), a new table is allocated, all its keys are rehashed using the new hash table size, and corresponding values are stored in the new hash table. The new table size is prime and is approximately double the previous table size. Although rehashing does take time, the results in Chapter 5 will show that the performance of the generator is not affected significantly. That is due to the fact that the time spent rehashing is small compared to the overall execution time. Because of rehashing, the number of states generated is not limited by any predefined hash table size.

Queue

The queue data structure is used to store the set of unexplored states. The data structure for storing the set of unexplored states was much easier to develop than the data structure for the set of generated states, since no searches or lookups are required, only insertions and deletions. A natural structure that has a constant insertion and deletion time is a special type of list named a queue. A queue was chosen not just because of the constant insertion and deletion times, but also because states are added to the end of the queue and removed from the front of the queue. By using this insertion and deletion policy, the state space is generated in a breadth-first manner.

An interesting modification used in the state-space generator implementation of the queue is dynamic resizing. As with the hash table, there is no *a priori* information about the total number of states in the state-space of the model, so it is necessary that it be possible to change the queue size during state generation. In particular, our implementation assumes a fixed size to start. If this size is exceeded, a new queue structure is created, and the states in the old structure are copied to the new structure. The number of times a new structure needs to be generated is minimized by doubling the queue size each time a new structure is produced.

4.2.3. Deterministically distributed actions

Three types of actions are supported in the Möbius state-space generator: exponentially and deterministically distributed timed actions and instantaneous actions.

Chapter 3 presented an algorithm that can directly be used for models' with exponentially distributed and instantaneous actions. Special attention is needed, however, to support deterministically distributed actions. In particular, when a model contains deterministic actions, the algorithm presented in Chapter 3 must be modified in a manner similar to that in [10] to generate the state space. More specifically, in this instance, the state-space generator generates a Markov regenerative process. To produce this process, modifications were needed in both the data structures and the output file. In particular, a new list data structure was added that keeps track of every state of the process, which enables a deterministic action. The generator uses this list when writing the generated process to disk. The format of the state-transition rate matrix file for a Markov regenerative process is similar, but not identical, to that for Markov processes. When an action is deterministic, the probabilities (rather than rates, as with exponentially distributed actions) to the next tangible state are written to disk. Each of these probabilities is appended with a minus sign to signal to the numerical solver that it is a probability instead of a rate. Additionally, a new file is produced containing the information about which deterministic action (if these is more than one) was enabled in a state and its associated deterministic parameter. Appendix B explains these files in more detail.

4.3. Output Formatting

The third and final stage of execution in the state-space generator is output formatting. In this stage, the files containing the state-transition rate matrix and reward file that were written during the state and reward generation stage are rewritten to the format specified by the user of the state-space generator. The state generator application supports six file formats as shown in Table 1. See Appendix B for more specifics on these output formats.

The importance of supporting multiple file formats lies within the numerical solution techniques that are used to solve the generated models. In particular, different numerical techniques require different types of access to the state-transition rate matrix (e.g., row-only, column-only, or both row and column) to obtain a solution. The choice of an efficient data structure thus depends on the particular solution method employed. Because of this, many different solvers will go as far as converting the input state-transition rate matrix from

Table 1: State-Space Generator File Formats

ASCII Row Format
Binary Row Format
ASCII Column Format
Binary Column Format
ASCII Möbius Format
Binary Möbius Format

its written form to a new form for solution.

The state-space generator has tried to provide many different file formats to avoid, as much as possible, the need for the conversion used by some numerical solvers. In addition to the different forms of the state-transition rate matrix, both ASCII and binary file types are available. ASCII files can be used when the files are needed over multiple platforms having different byte orderings. Binary files are available for single-platform use and can significantly decrease the amount of time needed to read in the file from disk.

The three formats for the state-transition rate matrix that the state-space generator supports are row format, column format, and the Möbius format. A file in *row format* contains the state-transition rate matrix in a form such that for every state, all outgoing states and rates are listed. Similarly, a file in *column format* lists the state-transition rate matrix such that for every state all incoming states and rates are listed. The *Möbius file format* also lists the state-transition rate matrix in column format, but adds three pieces of information: the maximum rate in the system, the total number of outgoing transitions, and for every state, the sum of all outgoing rates. These additions allow the use of popular numerical solvers, such as the successive-over-relaxation steady-state solver, that need to pass through a data file twice in row or column format, to pass through the file once.

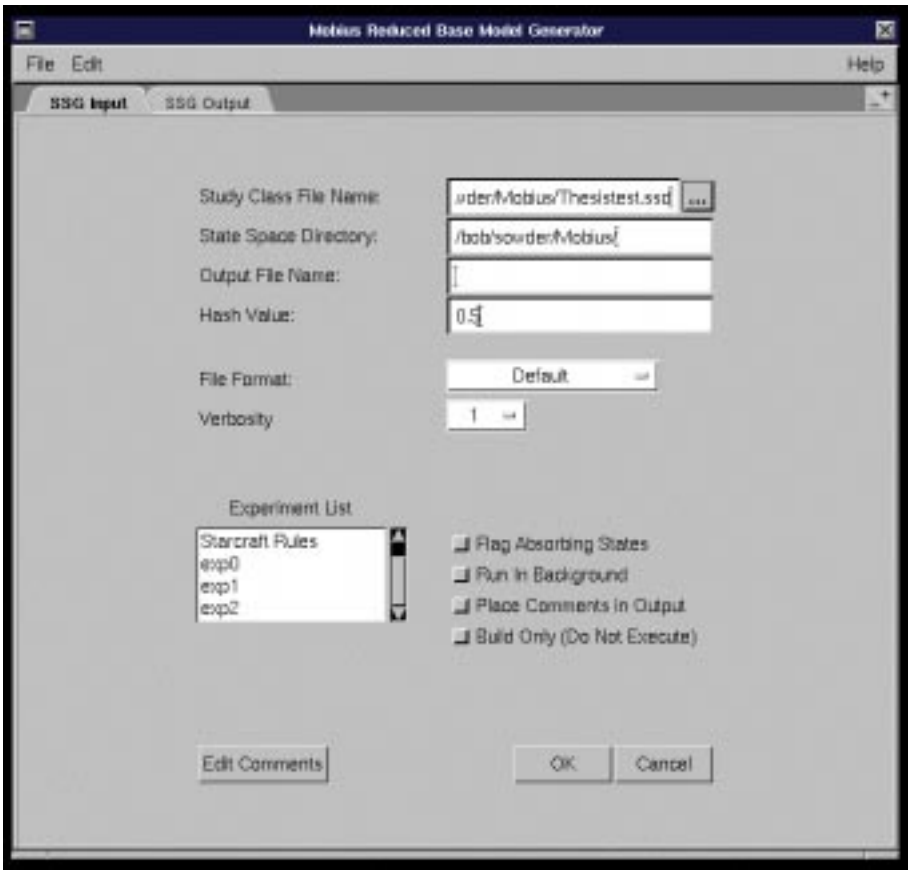

Figure 5: State-Space Generator Input

4.4. User Interface

Along with the underlying C++ implementation of the state-space generator in the Möbius tool, a Java interface was developed. The interface provides a simple and easy-to-use gateway to the state generation routine. The interface also provides an assortment of useful debugging options, execution options, file format options, and state generation information, as shown in Figure 5. The specific functionality of these options and more detail about the user interface is explained in Appendix A.

In addition to useful debugging and model-execution viewing features, the state generator interface displays the process of state generation during the generator's execution, as shown in Figure 6. To do this, the state-space generator process communicates the current

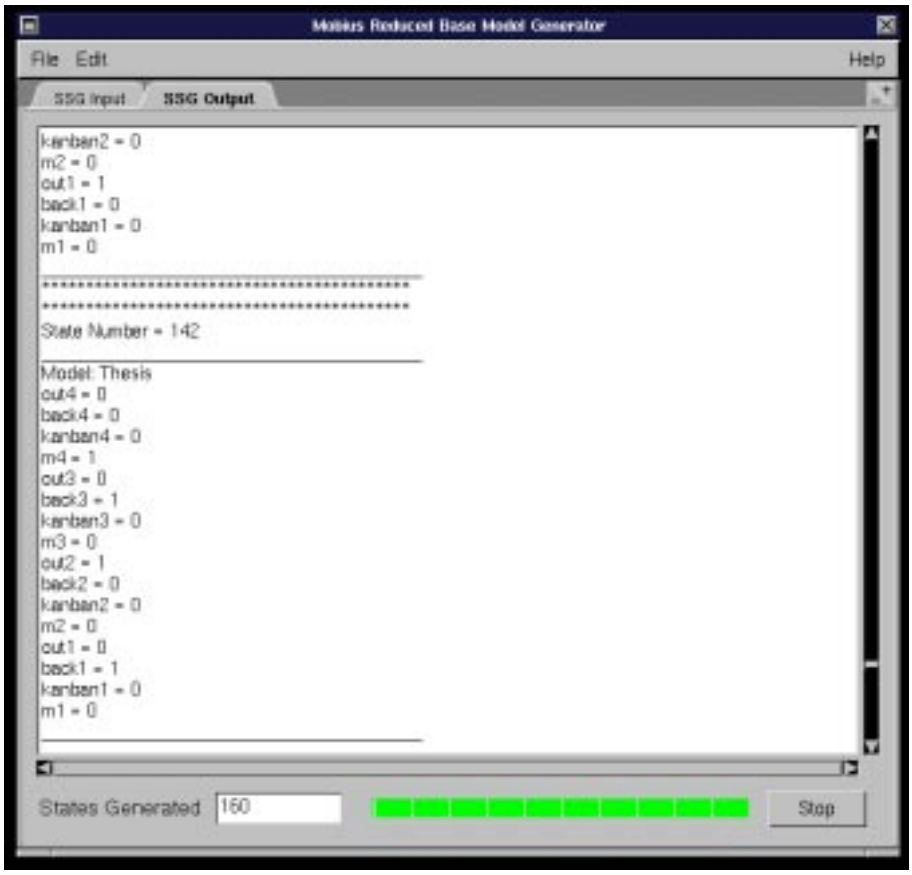

Figure 6: State-Space Generator Output

number of states generated to the Java interface as states are generated. Additionally, the interface allows for termination of the state-space generator process during execution. These features are useful for a modeler when executing, debugging, and analyzing the state-space of a model.

5. RESULTS AND COMPARISONS

In this chapter, we compare the space/time performance results of the state-space generator we developed with the state-space generator in *UltraSAN*. The main focus is on comparing their relative memory usage and total CPU time usage. While the particular state spaces generated were not important to this comparison, they were compared and found to be identical. Both generators were executed on an HP C160 workstation using a 160-MHz PA-RISC 2.0 processor. This workstation is equipped with 768 MB of main memory and 1 GB of virtual memory, and is directly connected to fast-wide SCSI disks. During result gathering, the systems had insignificant additional loads from other users. Two models were investigated: the Kanban model consisting of a single atomic model, and the faulty processor model. Each model had one state-based reward variable defined on it.

5.1. Kanban Model

The first model used for these comparisons is one previously used by Ciardo and Tilgner [12] to illustrate a Kronecker-based approach to generalized stochastic Petri nets. The model, referred to as the Kanban model, represents a simple factory production line. The model is divided into four stages. At each stage, an object either completes, moves to another stage, or returns to be worked on again. Objects leaving stage one may enter either stage two or stage three. Objects are recombined in the final stage, after proceeding through stage two or three. A SAN representation of this model is shown in Figure 7. The capacity of each state within the model is governed by the value of the Kanban places. By modifying the value of these places using global variables, we can observe how varying the capacity of the stages affects the overall throughput of the system.

We begin the investigation of the state-space generator performance on this model by varying the number of tokens in the Kanban places between one and six. This is done using a global variable within the model. Since there are four Kanban places in the model, this global variable also varies the total number of tokens in the model between four and twenty-four in

steps of four tokens. Increasing the number of tokens in the model increases the total number of states in the model dramatically, as shown in Table 2. As expected, increasing the number of states greatly increases the amount of memory the state-space generator needs to store the states of the model, and the time spent storing and checking generated states. This effect is shown in Figures 8 and 9.

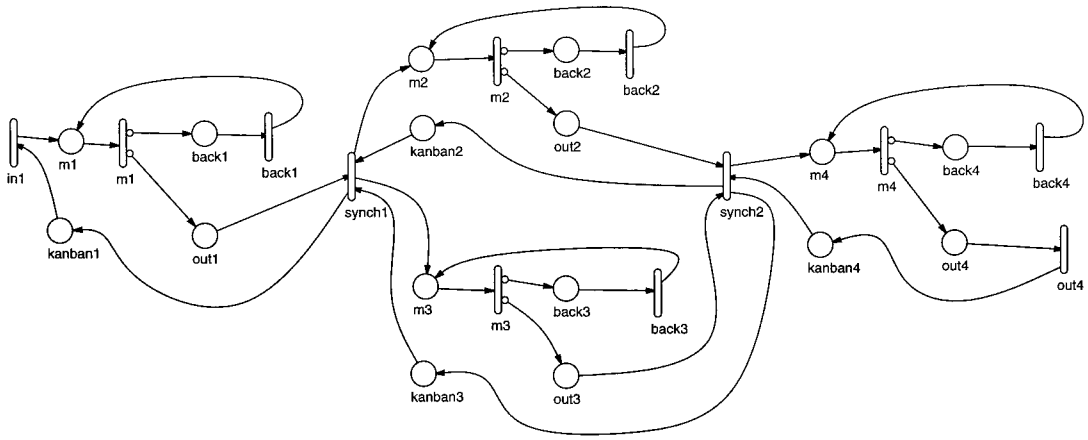

Figure 7: Kanban Model

Table 2: Number of States Produced by Kanban Model

Global Variable Value N, Initial Number of Tokens in Kanban Places	Number of States Generated
1	160
2	4,600
3	58,400
4	454,475
5	2,546,432
6	11,261,376

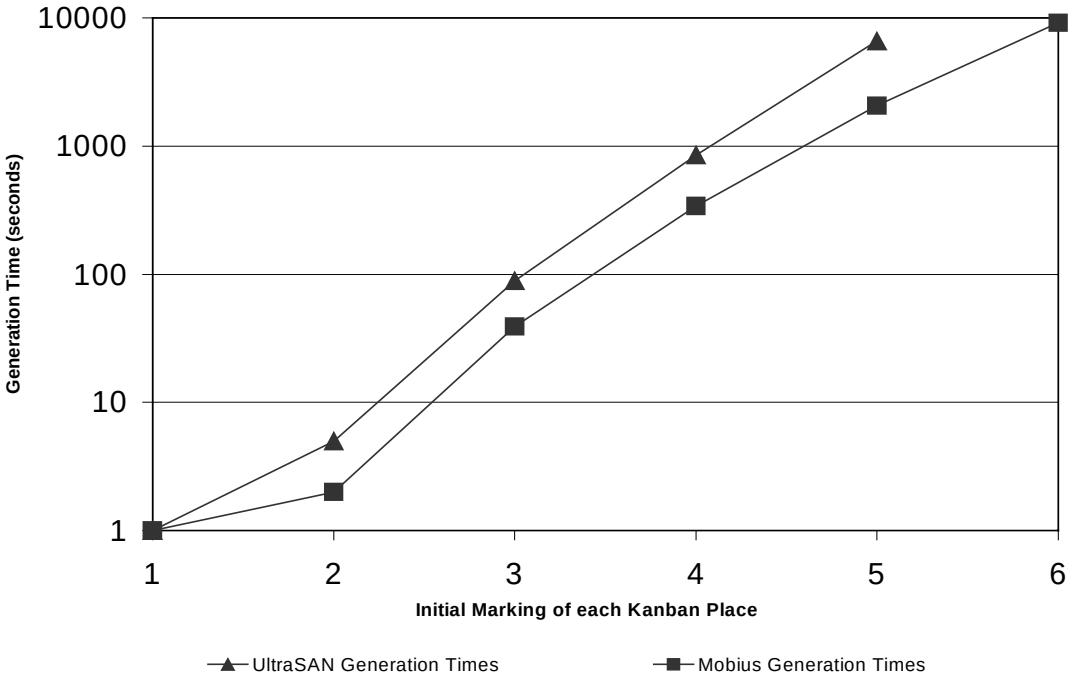

Figure 8: Generation Time Plot for Kanban Model

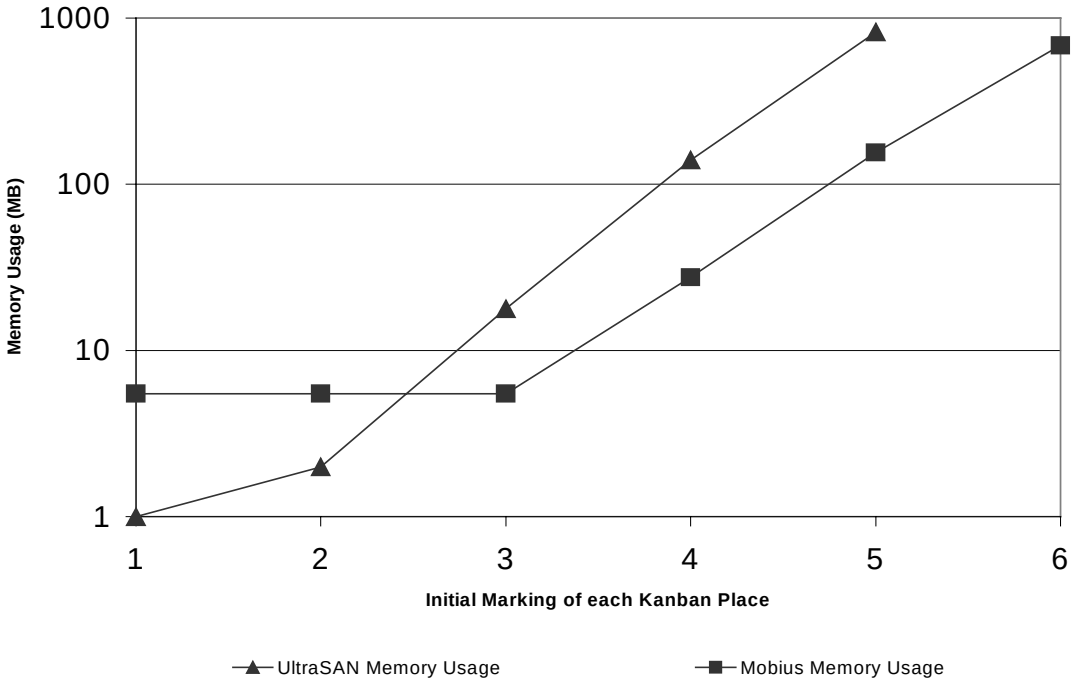

Figure 9: Memory Usage Plot for Kanban Model

As seen in these figures, the Möbius state-space generator provides a significant performance improvement over the *UltraSAN* state-space generator for both memory usage and total generation time. On average, the Möbius state-space generator was about twice as fast as the *UltraSAN* generator. For the cases with three to five initial tokens in the Kanban places, the Möbius generator needed, on the average, one-quarter the memory needed to generate the same number of states using *UltraSAN*. When the initial place markings of the Kanban places were 1 and 2, the Möbius generator used more memory than *UltraSAN*, due to the initial amount of memory allocated by the Möbius state-space generator. Finally, note that the Möbius state-space generator was able to produce the state space for the Kanban model when the initial number of tokens in the Kanban places was 6. The state space produced had 11,261,376 states. Extrapolating from the *UltraSAN* results for lesser initial markings, we believe the *UltraSAN* state generator would need approximately 4.5 GB of memory for this configuration, while the Möbius state-space generator needed 684 MB.

Another important comparison is the difference in the number of bytes per state and seconds per state needed by the two state-space generators. The results of these comparisons are shown in Figures 10 and 11. Figure 10 shows the number of states generated per second as the initial number of tokens in the Kanban places is varied from 1 to 6. After the initial time transient associated with the allocation of the initial block of memory, the states-per-second value approaches a constant value for this model. One reason a constant value is approached for the Kanban model is that the amount of time needed to execute the hash function stays the same as the number of states grows, since the number of memory values used as input into the hash function remains the same. Additionally, for this model, few collisions in the hash table occur, so that on average new states are inserted into the hash table in constant time. Figure 11 shows the number of bytes of memory used per state when processing various Kanban models. This plot shows that when the amount of memory initially allocated becomes small compared to the total memory usage, the curve approaches a constant value of 64 bytes per state, with the amount of memory needed for storage of individual states being 32 bytes.

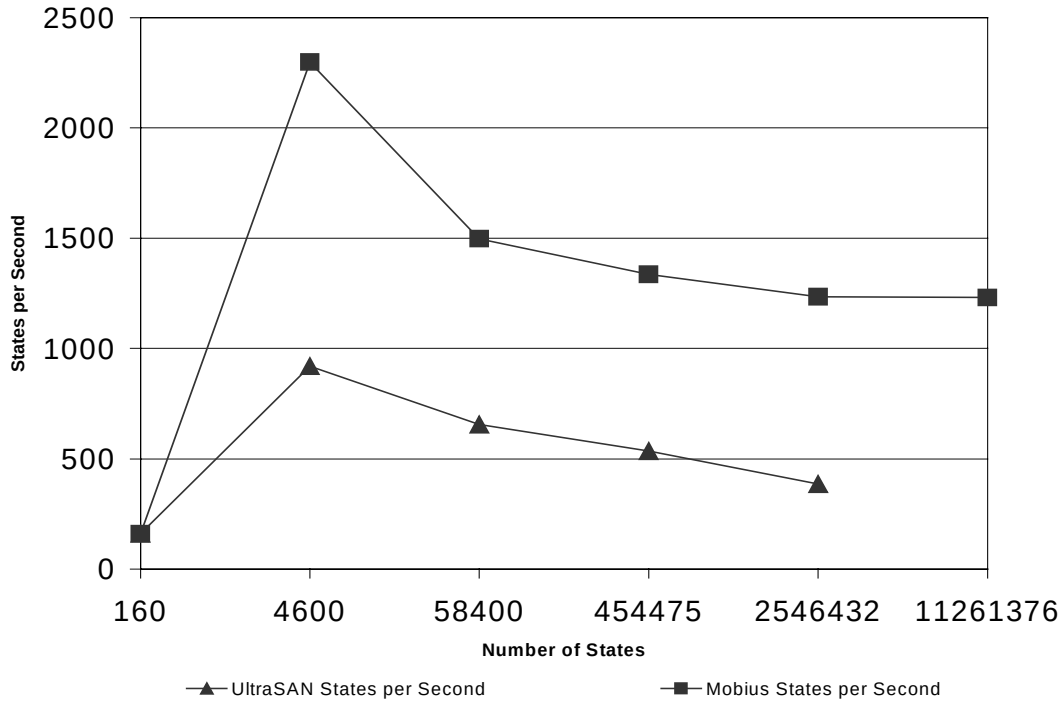

Figure 10: States per Second for Kanban Model

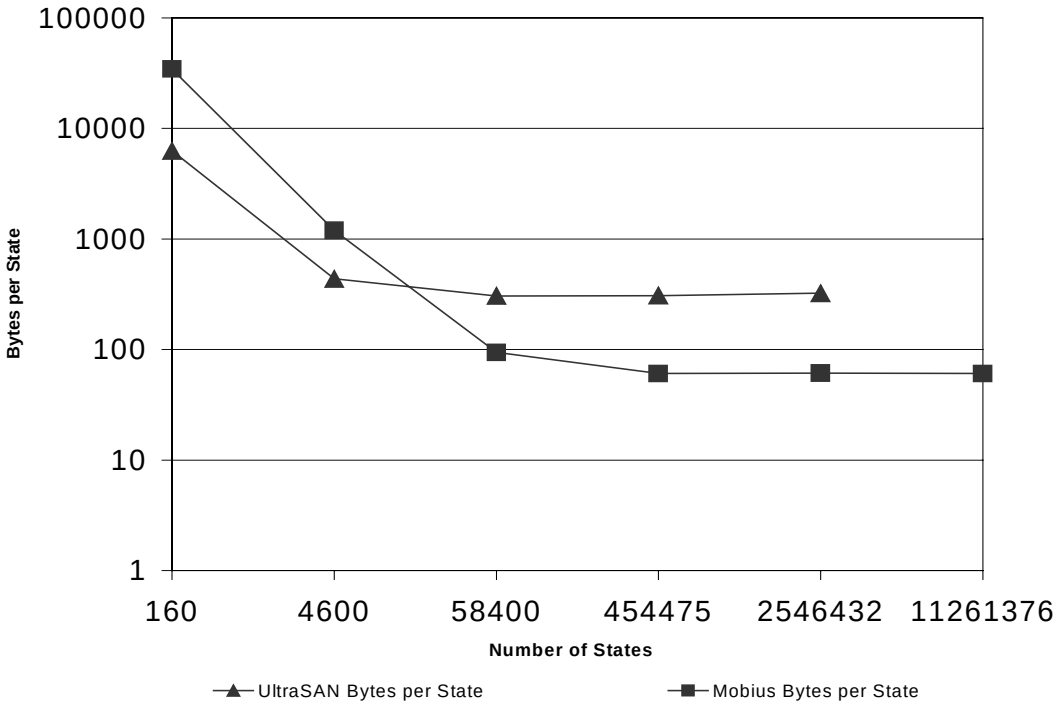

Figure 11: Bytes per State Needed for Kanban Model

5.2. Faulty Processor Model

The “faulty processor” model, found in [3], was also used to test the Möbius state-space generator. In this model, tasks for the faulty processor arrive as a Poisson process to a queue of certain capacity. If the queue is full, the task is rejected. The processor removes tasks from the queue on a FIFO basis. The processor can process at most two tasks at a time. If one task is processed at a time, correct processing is guaranteed. If two tasks are processed at a time, there is a chance of a processing error. A task that is processed incorrectly remains in the processor for processing. Figures 12 and 13 show the SAN representation of the processor itself and the job arrival process, respectively.

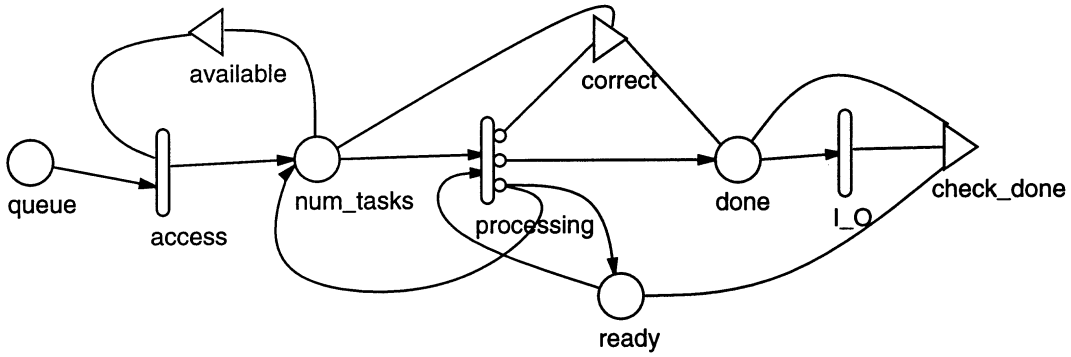

Figure 12: Faulty Processor Model

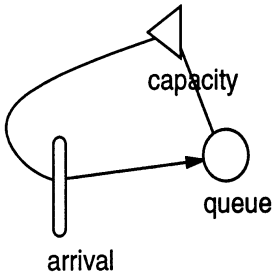

Figure 13: Faulty Processor Job Arrival Model

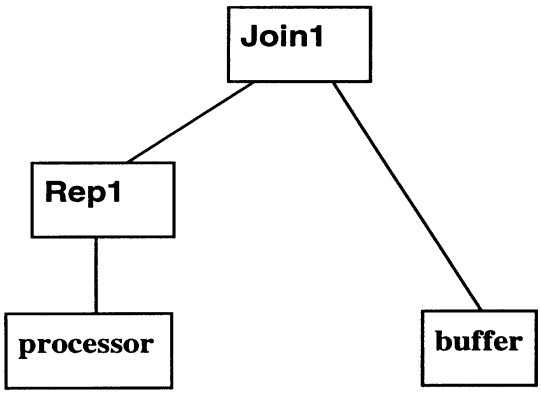

Figure 14: Faulty Processor Composed Model

The faulty processor model was used to test the scalability of the state-space generator, since this model can be used with the composed model editor in *UltraSAN*. The replicate/join formalism is one example of a composed model constructor available in the Möbius tool. The composed model used in this experiment is shown in Figure 14. By jointly varying the size of the buffer (controlled by global variable "size") and the number of processors (controlled by global variable "num_processors"), we can generate a series of models with approximately the same state-space sizes, but with different numbers of replications. These results can be used to study the performance impact on both state generator implementations, of an increase in the amount of memory needed to store. The global variable settings and resulting model characteristics are shown in Table 3.

Table 3: Composed Faulty Processor Model Settings and Characteristics

Global Variable num_processors	Global Variable size	States Generated	State Size (bytes)
1	20,000	180,009	9
2	4000	180,045	15
3	1090	180,015	21
4	362	179,685	28
5	138	178,893	35

The performance results for this model were compared to the *UltraSAN* state-space generator. These results are listed in Table 4. As with the Kanban Test model, the results are excellent, showing an average speedup of 1.5 times while using approximately one-quarter the memory needed to generate the same state space in *UltraSAN*.

The main reason this model was chosen was to investigate states generated per second and bytes per state when state size increases while the number of states is kept approximately constant. As expected, both implementations show a nearly linear increase in bytes per state as the state size increases, as shown in Figure 15, since the storage of the number of places in the model, and hence the state size, grows linearly with the number of processor submodels. The decreasing slope of the curve is due to the number of states for 4 and 5 processors being slightly smaller than for 1, 2, and 3 processors. Note that the Möbius state-space generator requires much less overall memory. In particular, for this model, both the absolute memory per state and the rate of growth of memory per state are less for the Möbius generator than the *UltraSAN*. Specifically, the Möbius state-space generator needs approximately 2.9 bytes of storage for every byte in the state size as compared to 13.6 bytes needed by *UltraSAN*. As seen in Figure 16, the states generated per second by both generators decrease as the state size increases. In the case of the Möbius state-space generator, this decrease in state generation rate can be attributed to the extra computation time needed to execute the hash function for larger state sizes. Therefore, a larger state size will cause more computations in the algorithm and thus increase the time needed to generate the state space of a model.

Table 4: Performance Results of Composed Faulty Processor Model

Global Variable num_processors	Global Variable size	Möbius State-Space Generator		UltraSAN State-Space Generator	
		Time (seconds)	Memory (MB)	Time (seconds)	Memory (MB)
1	20000	33	7.6	67	34.2
2	4000	73	8.5	116	39.8
3	1090	127	9.2	177	44.3
4	362	191	10.9	253	47.8
5	138	268	12.4	353	50.7

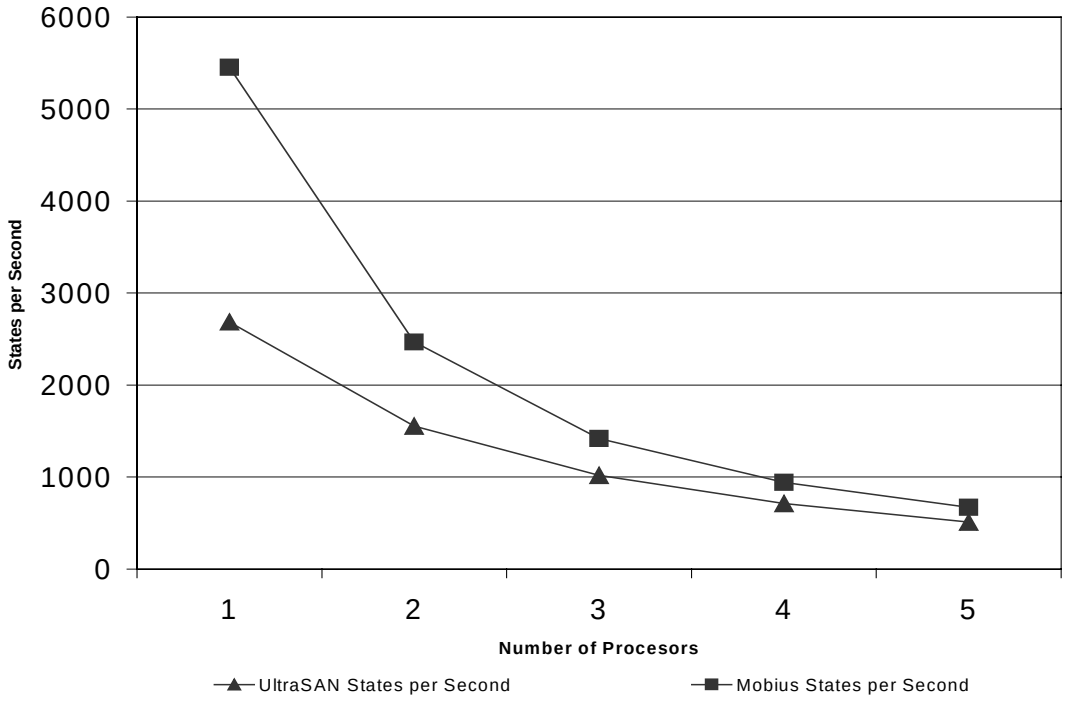

Figure 15: States Generated per Second for Composed Faulty Processor Model

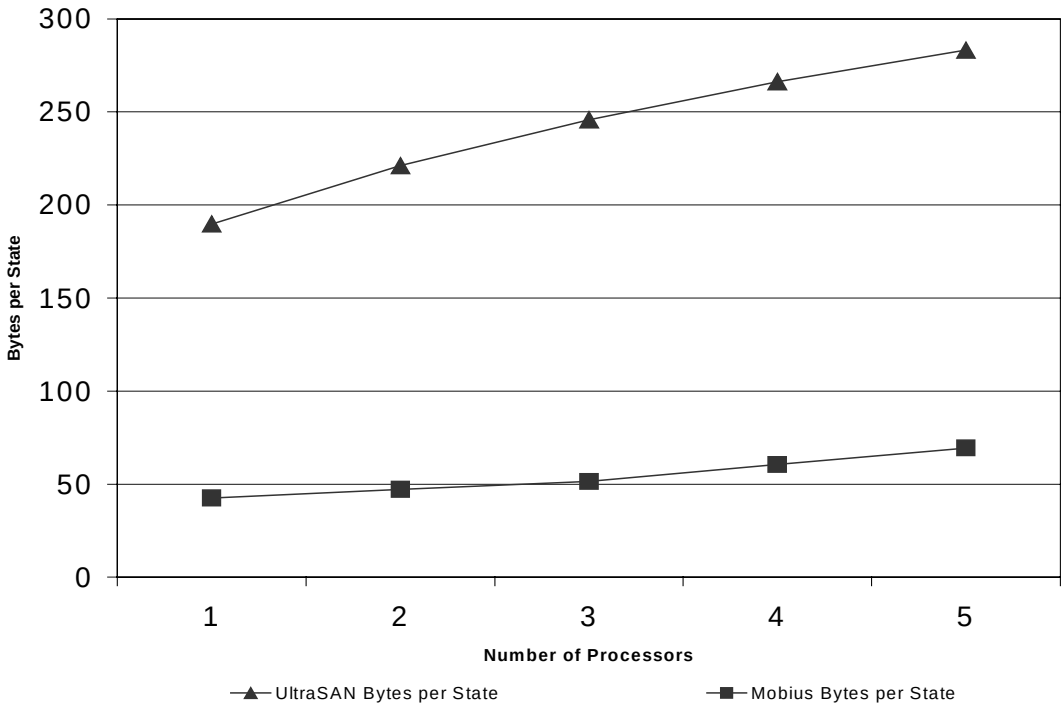

Figure 16: Bytes per State Needed for Composed Faulty Processor Model

As evidenced by the results for the Kanban and faulty processor models, the increased generality and abstraction of the Möbius tool do not decrease its space and time performance. In fact, Möbius outperforms *UltraSAN's* state-space generator in both dimensions. Of particular importance is the decreased memory usage, which will allow larger state spaces to be generated with the same amount of memory. By using innovative data structures and an optimized formalism-generic algorithm, we have been able to produce a full-featured, formalism-independent generation engine

6. CONCLUSION AND FUTURE RESEARCH

The objective of the work presented here was to develop a formalism-generic state-space generation algorithm that supports multiple action distribution types and also performs well by reducing the amount of time and memory needed for state generation. These objectives were achieved by:

a) Creating a state-space generation algorithm that is formalism-generic and that uses the methods provided by the Möbius tool.
b) Developing efficient data structures and code optimizations.
c) Implementing a state-space generator that supports exponential and deterministic timed actions as well as instantaneous actions. The instantaneous actions use the Möbius execution policy to decide what to execute during multiple enablings.
d) Supporting the state-space generation algorithm by designing a graphical user interface with useful debugging, analysis, and output features.
e) Implementing multiple output format types for the state-transition rate matrix, including the new Möbius format that speeds execution of numerical solvers.
f) Optimizing the performance characteristics of the formalism-generic state-space generator to exceed those of comparable formalism specific tools.

The utility of the Möbius state-space generator is shown by the performance results of the previous chapter. In every experiment for both models examined, the state-space generator substantially outperformed the *UltraSAN* state-space generator using identical models. In the future, the Möbius state-space generator can further be used to show performance results for multiformalism state-space generation.

Additional areas of research and additions could be explored. Specific to the state generation routine itself, additional data structure modifications could be implemented to combine the queue and hash table by using a flag on the hash table to signal whether a state is generated or unexplored. This implementation may save a significant amount of memory but would add additional complexity.

On-the-fly solution methods for transient solutions, another area of research, are currently being researched and are used to solve large models that have state spaces too large to fit in memory. For example, certain reliability models can be solved using this method. The Möbius state-space generator supports these new solution methods by allowing the designer to derive, using the C++ polymorphism properties, a new state generation routine. Thus, the designer could still use all the functionality in the state-space generator and just add the new methods to solve for the transition probability vector to determine transient measures, while at the same time generating only the states necessary for solution.

Another area of future research is phase-type distributions in state-space generation. We have performed research on this subject, including overview reading and a prototype implementation. We believe that this area of research may make it possible to model empirical data with a phase-type distribution, and to generate the state space of the model containing this distribution. The implementation of the Möbius state-space generator provides an easy and extensible interface for these future areas of research.

APPENDIX A: STATE-SPACE GENERATOR USER'S MANUAL

A.1. Möbius State-Space Generation

Before any of the analytic solvers may be used, the Möbius state-space generator must be executed to produce the state-transition rate matrix of a model's stochastic process. The Möbius state-space generator's output, which the solvers subsequently use as an input file, consists of two files for Markov processes and three files for Markov regenerative processes. The format of these files is explained in Appendix B. Once the Möbius state-space generator is installed, it may be selected from the "Solver" menu on the Möbius control panel. When a new state-space generator is created, the user is prompted for the name of the study file to open as the top-level model. An interface is then presented, which makes use of two tabs. The first tab is used to set input options used during state-space generation. The second tab shows the progress of state generation.

A.2. Installing the Möbius State-Space Generator Module

To install the Möbius state-space generator into the Möbius menu on the control panel, select "Add Module" from the "Modules" menu on the control panel, and fill in the entries as shown in Figure 17.

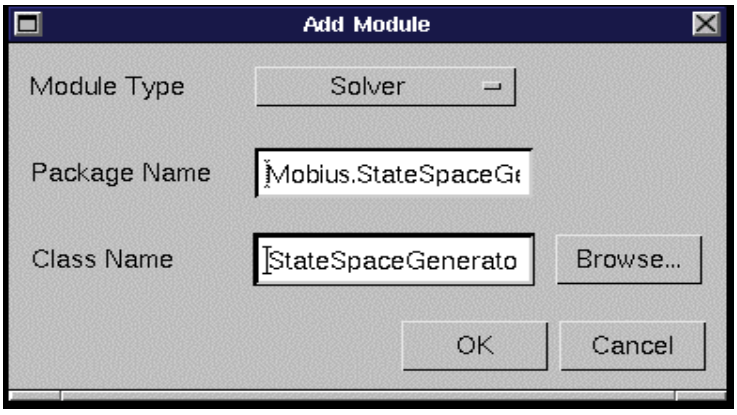

Figure 17: Adding the State-Space Generator Module

The entry for Package Name is `Mobius.State.SpaceGenerator`. The entry for Class Name is `StateSpaceGeneratorInterface.class`.

In order to use the state-space generator, the user must be able to remotely log into a computer and run Java with the proper `CLASSPATH` settings to include the Möbius classes. This setting may be set either for an entire system or on a per-user basis using startup scripts. To test this ability, execute the following command from a terminal prompt:

```
<remote shell command> <system name> java Arch m o
```

This should return the system architecture and operating system of the remote system.

The proper command-line Java must also be used to allow the state-space generator to operate reliably. In particular, the maximum native stack size and maximum Java heap size should be specified. The following startup line is recommended:

```
java -ss1M -mx64M StartMobius
```

The maximum heap size option (`mx`) may be adjusted larger, but caution is advised in allocating a larger native state size (`ss`).

A.3. State-Space Generator Input

The editor is accessed from the File→New→Solver→StateSpaceGenerator menu item on the control panel. Once this command is executed, a file dialog will appear, in which a user must specify the file name for either a range or set study. Once a valid study is provided, the panel shown in Figure 18 is presented. This panel allows the specific parameters of the state-space generation to be specified and also displays the names of the experiments associated with the study.

State-space generation parameters include:

- **Study Class File Name.** The study class file name is the location of the saved study file for the current state-space generation.
- **State-space Directory.** The state-space directory is the directory in which the state-transition rate matrix file, the reward output files, and the generator message output file are saved. If a directory is not specified before the OK button is pressed, then the user will be prompted to enter a directory.

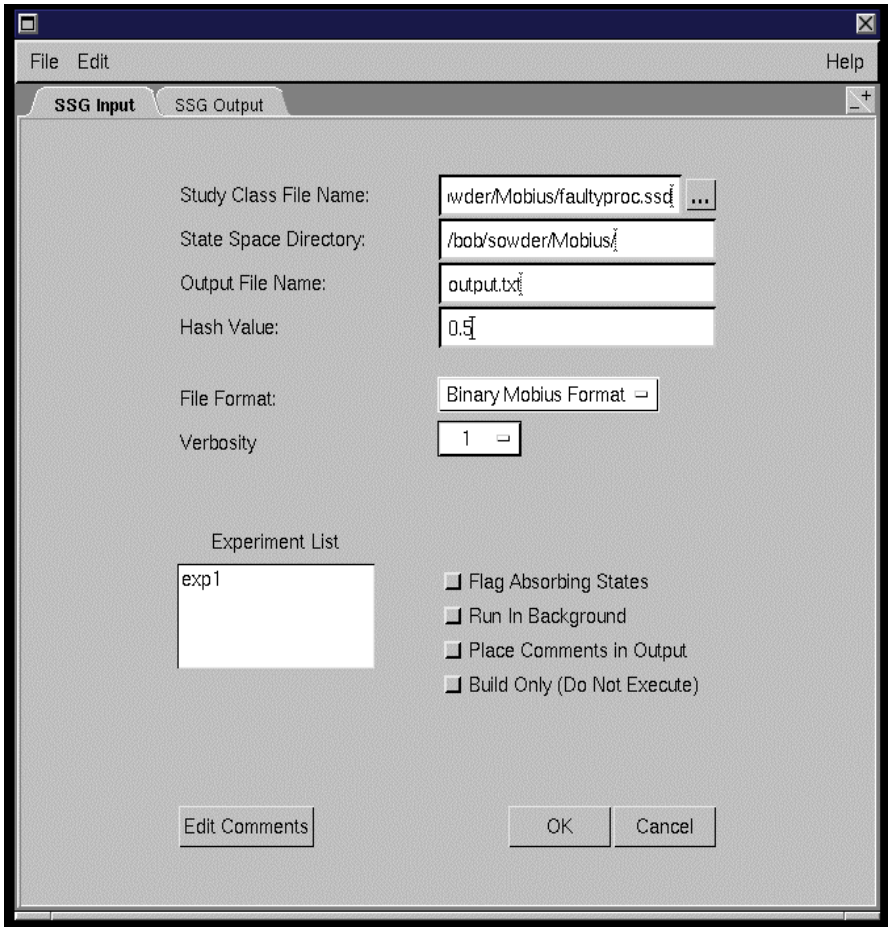

Figure 18: State-Space Generator Input Panel

- **Output File Name.** The output file name is the file used to store all the messages the state-space generator produces during execution.
- **Hash Value.** The hash value is the fraction of the hash table that is allowed to fill before a rehash occurs during execution of the state generation routine. This value can be between 0.1 and 0.9. A default value of 0.5 is used if no value is specified.
- **File Format.** The file format choice box sets the output format in which the state-transition rate matrix is written. The reward file is written in ASCII or binary depending on the format type specified. See Appendix B for a detailed description of these formats.
- **Verbosity.** The verbosity choice box allows various levels of output from the state-space generator. Allowable values are as follows:

  None: No output information is provided.

1: Provides the state number and its state variable values for every state generated.

2: Provides information about the dynamic reallocation of the data structures used in the state-space generator.

3: Provides detailed output of every action enabled in a state, the resulting states, and their rates or parameters. The user must check the "Run in Background" checkbox to use this verbosity level.

- **Flag Absorbing States.** When this checkbox is selected, all states in which no actions are enabled will be displayed. Detection of absorbing states is useful for debugging the model.
- **Run in Background.** This checkbox allows the state-space generation process to be run in the background, independent of the Java interface. When this box is checked, no execution information is displayed on the state-space generator interface.
- **Place Comments in Output.** When this box is checked, the comments entered using the "edit comments button" are displayed in the generator message output file.
- **Build Only (Do Not Execute).** When this box is checked, an executable for the generation of the state space is created. The state space is not generated in this case.
- **Edit Comments.** This button produces a new frame in which comments can be entered and then are displayed when the Place Comments in Output box is checked.
- **OK.** This button begins the state-space generator execution.
- **Cancel.** This button closes the state-space generator interface.

A.4. State-Space Generator Output

The state-space generator output tab, shown in Figure 19, allows the user to stop the state generation and also displays execution information. When the user starts state

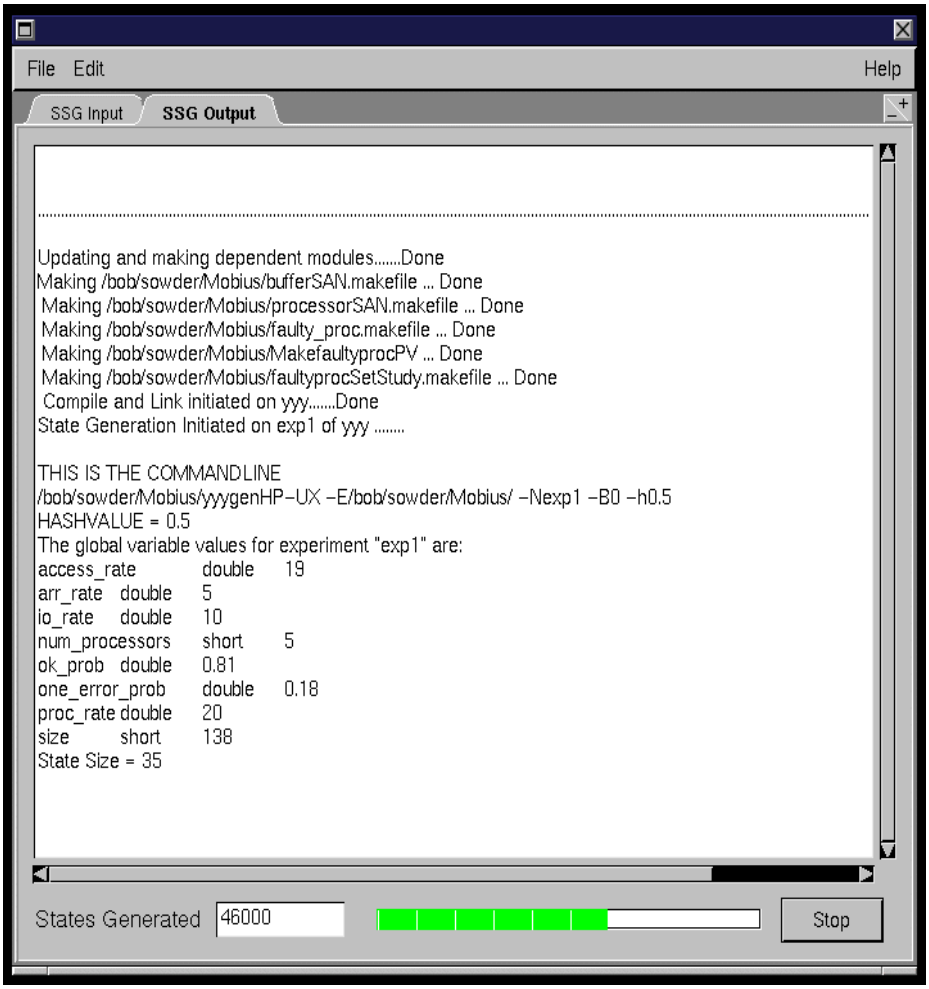

Figure 19: State-Space Generator Output Panel

generation by pressing the OK button, the interface validates all the entries that the user has selected to ensure proper operation. Any errors are displayed. After validation, the state-space generator compiles the source code to produce the state space generation executable. Then, while the generator executes, information about the progress of the generation process is displayed. Specifically, the progress bar increments for every one thousand states generated, and the "States Generated" display is updated after every one thousand states to show the number of states generated. The user may halt execution during state generation by pressing the Stop button.

A.5. State-Space Generator Tips and Tricks

The state-space generator is quite usable in its current form; however, some advice is necessary to achieve optimal performance and functionality.

- If your system has a large amount of memory and the expected state space is not excessively large, then set the hash value smaller than 0.5. An ideal value is 0.3; however, note that more memory is required for smaller values of the hash value.
- Make sure the model has a finite state space by putting limits on values in state variables.
- Realize that the number of states in a model is influenced by the introduction of action-based rewards. The generated number of states may therefore be higher than it would be without action-based reward variables.
- Explore all possibilities to reduce the state-space size, as a smaller state-space size speeds up all analytic solvers. In particular, try to model the system with the help of the composed model constructor, which may reduce the number of states.

APPENDIX B: STATE-SPACE GENERATOR OUTPUT FORMATS

This appendix gives the format of the output files that can be generated by the Möbius state-space generator. An understanding of these files may prove helpful in debugging models, and can also be used to link to a numerical solver. The generated stochastic process is contained in two or three files, depending on whether the model contains deterministic actions. Specifically, these files are: the file containing the state-transition rate matrix, the reward file, and the file used to store deterministic values if a Markov regenerative process is produced.

B.1. State-Transition Rate Output Formats

The files containing the state-transition rate matrix have details about the total number of states, the connectivity between states, and their rates (if all actions are exponential) or probabilities (if deterministic activities are used). The format of these files is explained in this section. Each explanation is presented in the form of a table of contents of the file, with each numbered item appearing on a new line starting from line No. 1 for each file type. The names of the files follow this format:

```
<"experiment name">.<extension>
```

Each section shows the specific extension for each format.

ASCII Row Format `<"experiment name">.arm`

1. A 1 in line 1.
2. The total number of states in the model under consideration.
3. State number of the current state.
4. The next possible state.
5. The rate from the current state to the state specified in item 4, if the associated action is exponential. If it is deterministic, a minus sign precedes the number in this line and should be interpreted as follows: The absolute value of the number in this line now gives the conditional probability that the deterministic action

completes going from the current state to the state specified in item 4. The time of the deterministic action is stored in the <"experiment name">.det file (see below).

6. Items 4 and 5 are repeated, each on a separate line, for all remaining next states and rates from the state in item 3.
7. A 0 to mark the end of the possible next states from the state in item 3.
8. Items 3 through 7 for all states in the model.

Binary Row Format <"experiment name">.brm

1. An ASCII 2 in line 1.

Items 2-8 are the same as for the ASCII row format, except the values are written in binary form.

ASCII Column Format <"experiment name">.acm

1. A 3 in line 1.
2. The total number of states in the model under consideration.
3. State number of the current state.
4. The next possible state.
5. The rate into the current state from the state specified in item 4, if the associated action is exponential. If it is deterministic, a minus sign precedes the number in this line and should be interpreted as follows: The absolute value of the number in this line now gives the conditional probability that the deterministic action completes going to the current state from the state specified in item 4. The time of the deterministic action is stored in the <"experiment name">.det file (see below).
6. Items 4 and 5 are repeated, each on a separate line, for all remaining next states and rates into the state in item 3.
7. A 0 to mark the end of the possible next states from the state in item 3.
8. Items 3 through 7 for all states in the model.

Binary Column Format `<"experiment name">.bcm`

1. An ASCII 4 in line 1.

Items 2-8 are the same as for the ASCII column format, except the values are written in binary format.

ASCII Möbius Format `<"experiment name">.amm`

1. A 5 in line 1.
2. The total number of states in the model under consideration.
3. The maximum departure rate in the system.
4. The total number of outgoing transitions in the model.
5. State number of the current state.
6. Number of non-self transitions into the state.
7. The total departure rate from the state.
8. The next possible incoming state.
9. The rate into the current state from the state specified in item 4, if the associated action is exponential. If it is deterministic, a minus sign precedes the number in this line and should be interpreted as follows: The absolute value of the number in this line now gives the conditional probability that the deterministic action completes going to the current state from the state specified in item 4. The time of the deterministic action is stored in the `<"experiment name">.det` file (see below).
10. Items 8 and 9 are repeated each, on a separate line, for all remaining next states and rates into the state in item 5.
11. Items 5 through 11 for all states in the model.

Binary Möbius Format `<"experiment name">.bmm`

1. An ASCII 5 in line 1.

Items 2-12 are the same as for the ASCII Möbius format, except the values are written in binary format.

B.2. Reward Variable File

The reward variable file for the Möbius state space generator contains all the information about the performance variables and the rewards (state-based and action-based) for all states. The file is written in either ASCII or binary depending on the corresponding type selected for the state-transition rate matrix associated with the reward file. The name of the reward file uses the following convention:

```
<"experiment name">.var
```

The format and contents of this file are as follows, with each numbered item appearing on a separate line, beginning with line No. 1.

1. Name of performance variable as it appears in the reward model editor, one per line for each performance variable defined in the model.
2. A state number.
3. Action-based reward for performance variable whose state is item 2.
4. State-based reward for performance variable whose state is item 2.
5. Items 3 and 4 for each of the performance variables defined, beginning with the first performance variable.
6. Items 2 through 5 for all states of the model.

B.3. Deterministic Parameter File

When the model has deterministic actions, the state-space generator will output a file containing the deterministic parameters. If the model does not contain any states in which the deterministic action is enabled, then the file will contain a single 1. The name of the file follows this convention:

```
<"experiment name">.det
```

The format for the rest of the file is as follows, with each enumerated item appearing on a new line starting from line No. 1.

1. A 1 to denote that the model has enabled deterministic actions.
2. The time for an enabled deterministic action.
3. All states in which a deterministic action can complete with the time value in item 2.
4. A 0 to mark the end of all possible states with the time value in item 2.
5. If more than one deterministic action exits, then repeat items 2 through 4 for each deterministic action.

REFERENCES

[1] A. M. Law and W. D. Kelton, *Simulation Modeling and Analysis*. St. Louis: McGraw-Hill, 1991.

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

[3] *UltraSAN User's Manual Version 3.0*, University of Illinois, 1995.

[4] F. Bause, P. Buchholz, and P. Kemper, "QPN-Tool: For the specification and analysis of hierarchically combined queueing Petri nets," *lecture notes in Quantitative Evaluation of Computing and Communication systems*, vol.. 977, Springer-Verlag, 1995, pp. 224-238.

[5] C. Lindemann, "DSPNexpress: A software package for the efficient solution of deterministic and stochastic Petri nets," *Performance Evaluation*, vol. 22, pp. 3-21, 1995.

[6] R. A. Sahner, K. S. Trivedi, and A. Puliafito, *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package*. Boston: Kluwer Academic Publishers, 1996.

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

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

[9] S. M. Ross, *Introduction to Probability Models.* New York: Academic Press, 1997.

[10] B. P. Shah, "Analytic solution of stochastic activity networks with exponential and deterministic activities," M.S. thesis, University of Illinois, Urbana, IL, 1998.

[11] M. A. Weiss, *Data Structures and Algorithm Analysis in C++*. New York: Benjamin/Cummings Publishing Company, 1994.

[12] G. Ciardo and M. Tilgner, “On the use of Kronecker operators for the solution of generalized stochastic Petri nets,” NASA Langley Research Center, ICASE Report #96-35 CR-198336, May 1996.