DISCRETE EVENT SIMULATION IN THE MÖBIUS MODELING FRAMEWORK

BY

ALEX LEE WILLIAMSON

B.E.E., University of Dayton, 1996

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

Discrete event simulation is an important method to determine the performance and dependability of many systems, including computer and communication networks. Whereas many analytic solution methods require models to conform to certain criteria or produce state spaces no larger than a computer system can manage, simulation has none of these limitations. This is the primary reason that simulation remains such a popular solution method despite the development of new analytic methods. There are many modeling packages available, most of which include simulators. However, each of these packages is able to create and solve models only within a single *formalism*. This is extremely limiting both to model construction and solution, but also to the advancement of new formalisms and solution engines.

The Möbius project solves these problems by providing an object-oriented, formalism-independent modeling environment. By utilizing the abstraction of this design, we are able to create a fast, efficient, cross-formalism simulation engine that provides solution power unavailable in other packages. This simulator is able to make use of distributed computing networks to manage multiple processors simultaneously. Also available is an assortment of built-in *execution policies* designed to simplify common modeling tasks and provide added modeling functionality.

To my family and friends.

ACKNOWLEDGMENTS

There are many people I would like to thank for their support both while I was here at the University of Illinois and throughout my academic career. First, I would like to thank Dr. William H. Sanders both for his technical guidance and for the opportunity to work with the Performability Engineering Research Group. I would also specifically like to thank Jay M. Doyle, Gerard Kavanaugh, and John Sowder for their friendship and support throughout the design and development of the Möbius project. Thanks also to Dan Deavours and Doug Obal for allowing me to tap their vast knowledge of system modeling and for their continual help and friendship. Thanks also to Jenny Applequist for her help in reviewing this thesis.

I would like to thank 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 would especially like to thank my family for their constant guidance and loving support. Thanks also to all my friends both here and abroad and to all those who have helped me along the way.

TABLE OF CONTENTS

Page

LIST OF FIGURES

1. INTRODUCTION

1.1. Möbius 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 design finalization. However, modern systems are often far too complex to prototype during the early development phases. For these systems, computer modeling and analysis is often a complementary addition to prototyping. Computer system modeling is a rapidly progressing field of research with many different approaches for representing a model. The current state of the art in model representation is the use of high-level modeling languages known as *formalisms*. These formalisms are often graphical in nature to allow easy system visualization and rapid model development. By analyzing models based on these formalisms, we can assess the system performance, dependability, and performability [1]. In recent years, advances in the processing power of the engineering computer workstation have made computer system modeling a valuable addition to system prototypes and often a viable alternative for early design analysis.

Many tools have been developed to simplify the construction and solution of computer models. Examples of these tools include *UltraSAN* [2], HiQPN [3], DEPEND [4], and DSPNexpress [5], among others. Although all of these tools provide simple interfaces for defining and solving models, these interfaces limit their means of model specification and solution. This is a result of the fact that each tool only supports a single modeling formalism and the associated solution methods. These interfaces can be extremely restrictive for complex models that are best represented using multiple formalisms. Likewise, research into the development of new formalisms and solution methods is hampered by the use of a single, fixed formalism. For instance, since each tool only supports a single formalism, development of new formalisms requires that a new modeling package be created.

Few have attempted to solve this single-formalism modeling problem by proposing a "universal modeling language" (see, for example [6]) that strives to define the most general and powerful formalism possible. The result of such a general formalism is that it omits information necessary for an analytic solution. To our knowledge, no one has attempted to create such a broad modeling language and retain the ability to detect when analytical solution methods are possible. The SHARPE modeling tool [7] allows multiformalism models; however, models in SHARPE interact in a limited way, exchanging results rather than sharing states or event types. This simple form of model interaction allows analytic solution methods specific to each formalism to operate on the model, but does not allow general solution methods to operate on a combined, multiformalism model.

Rather than attempting to create this universal modeling language, the Möbius modeling package addresses this problem by creating an extensible modeling environment. This environment allows model specification using a variety of modeling formalisms. Combinations of models using differing formalisms allow creation of new complex system models using the most appropriate formalism for each component. We call the models created through this combination *composed* models. Defining *reward variables* on these composed or stand-alone models allows the various analytic and simulation solution methods to solve for the behavioral characteristics of the model. Reward variables are used to evaluate a model based on organization and operational characteristics. Once these variables are defined, the model is *solvable*. Parameters within a model are manipulated by creating *studies*. These studies allow certain variables within a model to vary; each variation is called an *experiment*. Analytical or simulation-based solvers may then be used to analyze the derived solvable model.

We support multiple formalisms in the Möbius environment by identifying a set of characteristics and actions common to a variety of formalisms. By abstractly defining the properties of these components, we allow each formalism to implement the specific details independently. Formalism-independent components of the Möbius package, such as model composers, reward variable editors, study editors, and solvers, use these abstract interfaces to communicate. We include with the Möbius package a diverse and open-ended set of solution methods. These include analytic solvers for appropriate models and a simulator that is able to solve all models, regardless of the formalisms in which they are defined.

1.2. Möbius Simulation

Although analytic solution methods provide a variety of means to solve precisely for certain reward variables, these methods require that the model meet a stringent set of requirements. This often limits their solution techniques to a small subset of the models definable in the Möbius environment. The Möbius simulator provides a flexible and efficient alternative means of model solution for all single or composed models. Simulation provides a feasible means for solving many models, regardless of the model's structure or size. Only a small subset of models is unable to be simulated, such as, for example, models with rare events. By operating on the core set of abstract Möbius properties, we are able to construct a simulator that is able to execute any model defined by the Möbius tool.

The Möbius simulator is unique in its flexibility, power, and formalism-independent design. To our knowledge, no other simulation program attempts to create a solution method that spans formalisms. This allows for an easier interface for model designers, since the modeler may define each component using the formalism most appropriate to his or her design. The Möbius simulator also supports a variety of execution policies [8] to support a diverse modeling environment. These policies provide a foundation for new modeling formalisms and a basis for extensions of current formalisms.

Simulation speed in the Möbius simulator is gained by using distributed simulation. The Möbius simulator allows multiple processors to be distributed across single or multiple experiments. This enables several processing elements to work on independent simulations and to combine model observations. As computing power becomes more economical, arrays of networked high-performance workstations become increasingly available. The Möbius simulator is able to take advantage of this abundance of computing power by solving multiple simulations simultaneously and by using multiple processors to solve single simulations. Since each simulation runs independently, this produces a nearly linear overall solution speedup. Combined with the Möbius environment, the simulator allows for convenient model definition and solution.

1.3. Research Objectives

The goal of this thesis is to develop a formalism-independent distributed simulator for the Möbius modeling package. More specifically, the simulator developed will:

1. Be able to solve models derived from the basic components of the Möbius environment quickly and with results comparable to those of formalism-specific simulators.
2. Support distributed simulation such that a network of workstations may be used to obtain results from a single model efficiently.
3. Support a large assortment of execution policies, including reactivation, post-selection, pre-selection and the race policies of age memory, enabling memory, and resampling memory (see Section 2.3).
4. Provide a convenient and easy-to-use graphical interface from which to launch the simulation and monitor its progress.

We begin by explaining the details of the Möbius environment, discrete-event simulation, and model execution policies in Chapter 2. In Chapter 3, we discuss the details of the simulator and the methods by which the simulator interfaces with the Möbius environment. Chapter 4 presents performance results of the simulator and comparisons with the formalism-specific *UltraSAN* simulator [2]. We conclude, in Chapter 5, with a discussion of accomplishments and future research pertaining to the Möbius simulator. A users manual for the simulator is included in the appendix.

2. MODEL AND EXECUTION POLICY FOUNDATION

2.1. Formalism-Independent Model Specification

System evaluation using computer models has existed since the introduction of computerized analysis of product-form queuing networks in the 1950s. From these early models to more sophisticated formalisms, such as stochastic activity networks (SANs) [9], extended queuing networks [10], and queuing Petri nets [11], the field of system modeling has advanced in many ways. New languages for representing systems quickly and efficiently are under constant development. However, there is no one package that allows these new model formalisms to be combined and used together. Typically, a modeler is forced to choose a single modeling language to express his model. The Möbius package attempts to alleviate this problem by defining a set of generic classes from which many modeling formalisms may be derived. Once implemented using these classes, formalisms are open to the wide assortment of composition, reward variable specification, and solution methods available within the Möbius environment.

In order to create such a flexible system, the Möbius environment makes extensive use of object-oriented programming and design. This is accomplished through the C++ and Java programming languages. The abstraction that these languages provide allows a set of generic object base classes to be defined such that specific components of the system may be derived. By studying many formalisms currently in use, we have observed that most useful formalisms have elements that *hold state*, and elements that *affect state*. We define the state of the model to be the configuration of the system, expressed at a level of abstraction appropriate to specify its future behavior and capture desired performance and dependability measures. For instance, consider a model created to analyze a production line. The state of this model might be the number of components currently in production, the number of pieces in inventory, and the number of workers on the line. Of course, the exact representation of state is likely to change based on the desired characteristics under evaluation. The interaction between state-holding and state-affecting elements results in the model changing state. In the Möbius framework, the classes developed to portray the state-changing, or affecting, elements are called *actions*. Those that hold state are called *state variables*.

2.1.1. Actions

Actions in the Möbius framework define the model components that change the state of a model. The exact implementation of the state-change mechanisms, including details of the enabling functions, rate parameters, and interactions with the state of the model, is left to the specific formalism derivations. The base classes simply define a set of data members and methods that must be defined on the objects in order to allow the composition, reward, and solution methods to operate. For example, the derived stochastic activity network (SAN) [12] model formalism uses these actions as the base classes through which activities, input gates, and output gates are defined.

Actions change the state of the model by *firing*. This term refers to formalism-specific means by which actions and state variables interact to produce state changes. Solvers in the Möbius environment are unaware of the details of the formalism implementation by which state change is accomplished. However, interactions with the model are consistent across formalisms because of their derived structure. In order to fire, an action must be *enabled*. Just as a production line worker is able to work on a part only when one is available, an action is able to execute only if the state of the model meets the action's enabling specification. If the action is not enabled, it is *disabled*. The simulator also uses an object called an *event* to maintain a representation of an action. This representation is primarily used to record the time at which the action will fire. This allows the simulator to sort events such that the action with the earliest firing time can easily be extracted from a list. The list used to sort events is called the *future events list*.

One important property of actions in the Möbius environment is their use of "ranks" and "weights." These properties allow actions to be given distinct priority or probabilistic weighting over other actions when events are scheduled for the same time. This functionality is useful for handling conflicts between actions as priority is given to the set of actions with the highest rank. Probabilities using the weights of the actions are used to choose an element of the set for selection. A useful extension available as a result of these components is the creation of action “groups.” A *group* is a set of actions combined together for selection purposes, regardless of firing time. Groups themselves behave as actions, but selection of specific actions within the group may occur as a result of interactions of the model and the solvers. This action composition allows for policies to be developed that define when actions within groups are selected and the processes by which the selection occurs (see Section 2.3.1).

2.1.2. State variables

State variables within the Möbius environment are the basic units that are acted upon by the actions. The combination of all the values of the state variables in a model represents the state of the model. For example, the place components of a SAN model are represented using the state variable class. The derived class from these components may be as complicated as the formalism requires, from basic data types to complex structures. The model formalisms are used to describe the interaction of actions and state variables. This includes parameters such as how state variables are used to enable actions and how the actions act upon the state variables in order to modify the state of the model.

The most important notion of actions and state variables with respect to the solvers is that they encapsulate the specific interactions of the formalisms. By operating on these components, the state of the model is changed appropriately, no matter which formalism they represent. State changes may be used to traverse a possible sequence of states that may occur in the model or to explore all possible states that may be reached by a model. The former state change allows model simulation while the latter enables state space generation, for analytic solution methods. The exact mechanism the model uses to change state is encapsulated into the formalism. This level of abstraction is the key to creating model-independent solution methods. As long as the components of the formalism can be represented through derivations of these base components, the composition, reward, and solution methods may be utilized.

2.2. Discrete Event Simulation

A model is created by combining these actions and state variables such that they interact to represent a system. However, in order to observe characteristics of the model's operation, reward variables must be defined on the model. These variables accumulate observations based on functions of specific states of the model or firing of specific actions. Proper definition of these variables is essential for developing a model that accurately represents the system under consideration. Once the reward structure is created, the model may be solved by either analytical or simulation means. Analytic solvers typically require that a state space of the system be generated that includes all possible states of the model and transition rates between these states. From this state-transition-rate matrix, the behavior of the model may be determined. Although the accuracy of this solution method is comparable, if not numerically superior, to simulation, it is limited to models that conform to certain parameters and those for which the state space can be generated. Often simple models result in extremely large state spaces. Furthermore, the type of actions that can be considered is limited to those using exponential, deterministic, and Erlang distributions. Although widely accepted by the academic community, industry continues to prefer less restrictive means of solution, such as simulation. Thus, although the Möbius environment includes an extremely efficient state space generator [13], this is not always the best, or even a possible, means for solution.

The general concept behind stochastic simulation is that by executing a model through a sample state-change path, an observation of the reward variables is generated. By repeatedly executing the model, many statistically independent observations of the reward variables may be produced. Individually, each observation may not accurately represent the true nature of the system, but combined, statistical analysis of the observations provides a statistically significant estimate of the desired performance or dependability measures. An important requirement, which allows every execution to produce useful observations, is that each execution of the model uses a different random number seed. Because actions typically use samples from numerical distributions to determine the length of time before the action occurs, varying the random number seed allows for distinct, yet statistically independent observations. As the simulator acquires more observations of the reward variables, the estimated values of the variables become more exact and *confidence intervals* of the variables become more narrow. The confidence interval provides an insight into the statistical precision of the variables.

As shown in Figure 2.1, the general algorithm used for discrete event simulation is quite simple. For this algorithm, E represents the future events list, μ is the state of the model, EN_μ are the actions enabled in the state μ, and e_a is the event representing the execution of action a. The general process is to first generate a list of events for all actions enabled in the initial system state. Lines 3, 4, and 5 of the algorithm show this process. The earliest event on the future events list is then fired. An updated future events list is then generated by removal of actions that have become disabled from the list and addition of those that have become enabled. Lines 10 and 13 of the algorithm show the removal and addition processes respectively.

```
1     $E = 0$
      $\mu$ = INITIAL MARKING
      $\forall\, a \in EN_\mu$
          $e_a = GenerateEvent(a, \mu)$
5         $E = E \cup \{e_a\}$
      $while\ (E \neq 0)$
          $\mathrm{e_a} = Earliest(E)$
          $\mathrm{E = E - \{e_a\}}$
          $\mu' = FireEvent(\mathrm{a}, \mu)$
10        $\forall\, e_a \in E$
              $if\ (a \notin EN_{\mu'})$
                  $E = E - \{e_a\}$
          $\forall\, a \in EN_{\mu'}$
              $if\ (a \notin E)$
15                $e_a = GenerateEvent(a, \mu')$
                  $E = E \cup \{e_a\}$
      $end$
```

Figure 2.1: General Discrete Event Simulation Algorithm

The algorithm continues until either the model reaches an absorbing state, in which case there are no actions left to execute, or the end of the accumulation period for the reward variables is reached (this is not shown in the algorithm). Both steady state and terminating simulation use this general algorithm. A terminating simulation is used to estimate measures at particular times or over particular intervals of time. In this case, each execution proceeds until the reward variable collection interval is completed, records the observations, and resets the model to the original state to repeat the process with a different random number stream. Using the batch means method, a steady state simulation executes the model until an initial transient time period is passed. The length of this interval may be determined either by the modeler or by analysis of the model itself. Once this period is completed, the reward variables begin to accumulate observations. As each observation is generated, the value is recorded and the model proceeds with the next observation. Groups of observations are broken up into "batches" that are considered to be uncorrelated. At no point in the steady state simulation process is the state of the model returned to the original state. These two types of simulation allow for insights into different aspects of the model.

2.3. Execution Policies

The system model itself provides much of the functionality for simulation in the Möbius environment. This is required because the solvers must be able to access the model using formalism-independent references. As such, models within the system must be able to maintain their own state during the execution as well as update the state of the model through the firing of actions. The simulator maintains the progression of the model by organizing the firing of actions using a future events list. This data structure sorts the execution time of actions such that the next most recent occurrence of an action can quickly and easily be determined. The algorithm used to determine the precise interactions of an action and the modeling containing it is called the *execution policy* [8].

The Möbius simulator supports a variety of execution policies to provide a rich modeling environment. Currently, support is included for post- and preselection groups, race memory policies [8], and reactivation [9]. As discussed previously, the selection groups choose a member action for execution based upon a probabilistic selection. Race memory policies define the effects of the removal of an action from the future events list. Reactivation allows for a complex set of predicates and functions that determine when actions should choose new completion times.

2.3.1. Selection policies

Selection policies in Möbius define the way in which a representative action is selected from a group of actions at various points during the execution of a model. The selection policy depends on two properties of an action: rank and weight. Probabilistic selection is used to select an action from the set of enabled actions within the group with the highest rank. The weight of an action specifies the probabilistic bias given to an action. The preselection group stipulates that the selection is to occur with a group when any member of the group becomes enabled, thus enabling the entire group. The postselection group performs the selection process immediately before an action fires. These group selection policies provide the modeler with an extra degree of power for modeling complex systems. For instance, preselection may be used when modeling the traffic patterns of a global router system. The path an incoming data packet takes through the network may be determined by the status of the network upon arrival. Postselection groups are used specifically in the implementation of the Möbius SAN modeling formalism to create activities with cases.

2.3.2. Race memory policies

The race memory execution policies are used to define the effects of enabling and state recollection on the execution of an action. The term “race” is used to describe these policies because there is no predetermination or probabilistic selection of the order in which actions fire. The actions thus compete, or race, to determine which executes first. The race age memory policy provides a notion of past work to an action while the enabling policy completely ignores work done previously. Race resampling memory analyzes the effect of state memory on an action.

Age

The race age execution policy specifically identifies how memory of a partial completion affects the new enabling of previously disabled actions. This represents the situation when completion of an action does not occur each time it is enabled, although, work may be progressing towards completion during each enabling of the action. Thus, by recording the portion of work completed by an action previously enabled, a more exact estimate of the length of time necessary to complete can be developed when the action becomes enabled in the future. When a race age memory action becomes enabled, we cannot simply sample the numeric distribution for the time until completion; rather we must modify the sample to account for previous work completed by the action. This policy may be useful for production line models for which production may be interrupted for periods of time, with the components already in progress remaining partially completed when the production line is restarted.

Enabling

Unlike the age memory policy, the race enabling memory policy completely ignores the effects of past work on the completion times of newly enabled actions. This is the standard case by which most formalisms operate, including SANs. When an action using this policy becomes disabled before completion, any history of work completed is lost. If the action becomes enabled again before the model completes execution, the new sample of time before completion of the action is entirely determined by the numerical distribution. A simple example of where this policy is useful is the arrival of a customer into a service center. Customers either arrive completely or not at all; there is no memory of customers that almost arrive.

Resampling

The race resampling memory policy stipulates that the numeric distribution, from which the interval of time to completion is taken, be resampled with each state change of the model. Since the state of the actions using this execution policy is guaranteed to be independent of the history of the model, this execution policy allows for nonexponential distributions to behave with some of the properties of a memoryless distribution. This property allows for a larger set of models to be analyzed using the principles of the Markov process. An example of a simple model that might make use of this policy is a service center that gives no priority to customers. When a customer completes execution, the distribution of each customer left in the queue is resampled to determine who is next to receive service.

2.3.3. Reactivation

The reactivation execution policy provides functionality similar to that of the race resampling memory policy. The primary difference is that an action's ability to resample its distribution is based on a reactivation predicate and function. When an action is enabled, the reactivation predicate is evaluated. If the predicate is found to be true, the action then becomes *reactivatable*. If at any time while the reactivatable action is enabled the reactivation function becomes true, the distribution of the action is resampled. Thus, as long as the reactivation function remains false, a reactivatable action behaves as a typical race enabling memory action. Also, if the reactivation predicate is evaluated as false upon enabling, the state of the reactivation function is irrelevant and the action behaves as a race enabling memory action. This policy thus allows the action to conditionally switch between two differing execution policies.

The combination of these selection, race, and reactivation execution policies defines a set of algorithms that determine precisely how each action within a model operates. Although ultimately governed by the formalism design, each action within the Möbius framework may use separate execution policies. Since these policies deal with the time and work of an action, analytic solutions using these policies is not possible. However, since analytic solution methods require memoryless distributions, there is no need for such policies. By combining these individual action execution policies into the generic simulation algorithm of Figure 2.1, we generate an overall model execution policy. These action policies have also been referred to as an action's *work policy* [14] since they define interaction between an action and the work the action completes.

3. SIMULATION DESIGN OVERVIEW

3.1. Simulation Engine

The Möbius simulator is divided into two components, the C++ simulation engine and the Java graphical interface and simulation manager, as shown in Figure 3.1. This division between C++ and Java allows Möbius to run on a wide variety of computing platforms while still providing the computational power necessary for real-world modeling tasks. The simulation engine is designed as a simple mechanism to execute the model, collect batch observations, and send the observation to the Java simulation manager. The engine does not maintain statistics on the observations and thus must rely entirely on the manager for the stopping criteria. One of the benefits of this design is its ability to easily expand to a distributed simulation. Each simulation engine runs completely independent from other simulations; thus, by simultaneously launching several simulation engines, with different random number seeds, the manager is able to receive and calculate statistics on distinct, nonoverlapping batches from each engine.

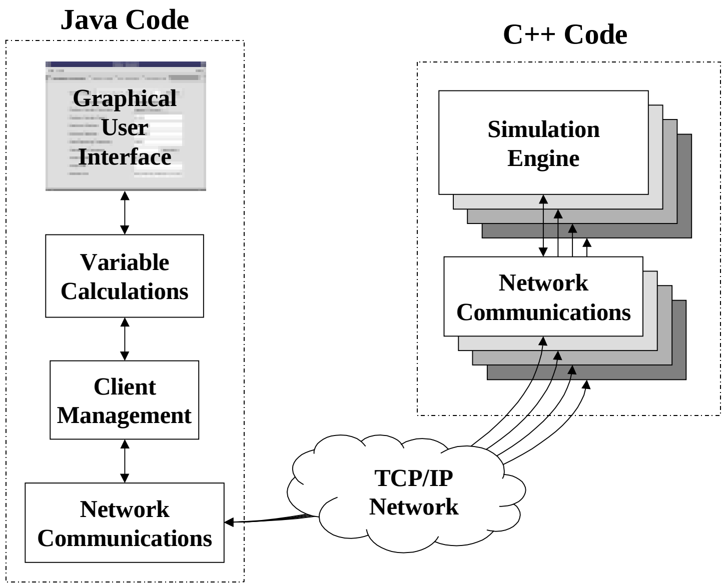


Figure 3.1: Simulator Architecture

The core of the simulation engine is designed as a single C++ class containing all of the necessary methods to execute a model. Although the abstraction and inheritance of C++ introduces a slight overhead for most compilers, efficient data structures and algorithms can effectively compensate for any losses. The class structure of C++ also provides a more extensible and reusable design. In addition to the primary class, the simulator generates classes to initialize data structures, manage future-events lists, and provide random number streams and distributions. As mentioned previously, the simulator also supports both terminating and steady-state simulation as well as a variety of execution policies. These policies include race age, race enabled, race resample, reactivation, preselection, and postselection. Figure 3.2 provides a diagram of the architecture and data flow within the simulation engine. The next sections discuss the details of these architectural components.

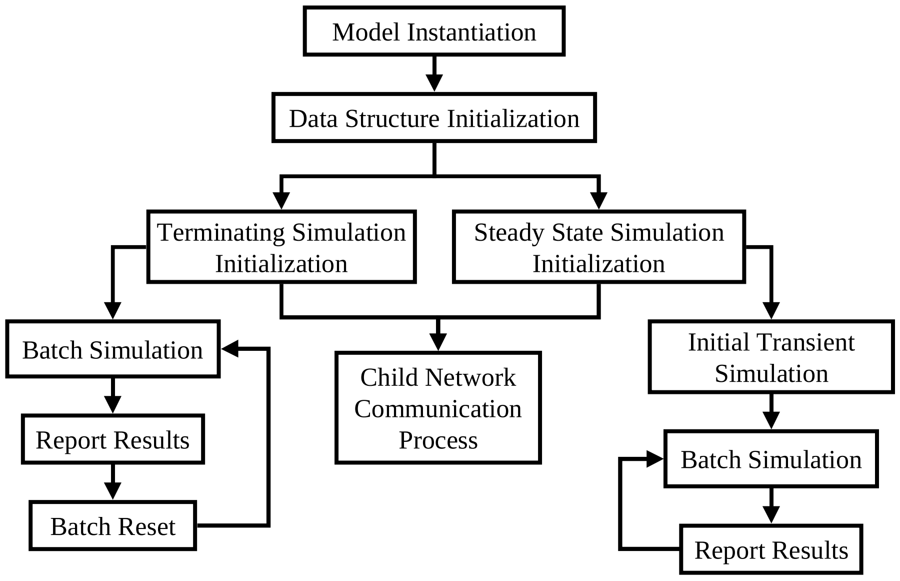

Model Instantiation
Data Structure Initialization
Terminating Simulation Initialization
Steady State Simulation Initialization
Batch Simulation
Child Network Communication Process
Initial Transient Simulation
Report Results
Batch Simulation
Batch Reset
Report Results

Figure 3.2: Simulator Engine Data Flow

3.1.1. Initialization

Prior to reaching the core of the simulation loop, the simulator must complete several stages of initialization. These steps consist of model instantiation, creation of data structures to simplify the state change and reward accumulation procedures, and construction of a child process to communicate with a manager. Since the primary purpose of the simulator is to generate model solutions, the top-level solvable model is the only model with which the simulator has direct interaction. The property of a model being solvable simply indicates that reward variables are defined on the model such that a solver (e.g., the simulator) may execute the model. This level of abstraction between the simulator and any specific formalism helps to isolate the simulator from specific model structures. In order to instantiate the solvable model, the *study* must be created and initialized to the proper experiment. The study is a structure that allows a single model to be analyzed using a variety of system configurations. This model manipulation is done using *global variables* within the model. These variables may be defined as the model is being designed and changed using the study to create different *experiments*, or test configurations for the model. To allow the code of the solvers to be static, the study is instantiated by a call to an external function. The study then instantiates a specific instance of the solvable model. This process allows the simulator, as well as other solvers, to function on a new model simply by linking the executable with a different set of model-specific libraries.

The next phase of initialization is the creation of action connectivity lists. These lists provide the simulator with the information necessary to determine which actions within a model may affect other actions. Typically, after an action fires in a model, a standard simulator checks every action in the model to determine its current enabling status. This can lead to a large simulation overhead that increases linearly with the size of a model. By using these connectivity lists, the simulator is able to greatly reduce the number of actions that are checked for change in status following each state change. This can allow for a significant performance increase in most models. To facilitate creation of these lists, each action in the model contains a list of state variables that affect the enabling of the action as well as a list containing the state variables whose state may be changed by the firing of the action.

For example, in a stochastic activity network, if the marking of a place affects the enabling function of an activity, then the place would appear in the activity's enabling list. Likewise, a place whose marking changes as a result of an activity firing would appear in the affected list of that action. Specific detail of the SAN implementation in the Möbius modeling framework may be found in [12]. Often enabling and affected lists overlap due to symmetries within the model.

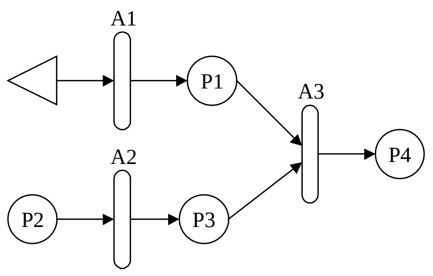
Through the transformation of the provided action-to-state-variable lists into a state-variable-to-action list, the desired connectivity can be extracted more easily. The algorithm for this conversion is shown in Figure 3.3. Recall that in the simulator, we consider a model to be a simple set of actions and state variables. We organize the state variables into a *StateVariableList*, indexed by the state variables of the model. Each *StateVariableList* element also contains action lists named *EnablingList* and *AffectedList*. These lists are used to represent the actions to which a state variable is "enabling to" and "affected by," respectively. Initially, these lists are empty. We also simplify the representation of an action to a set of three lists: *EnablingStateVariables, AffectedStateVariables,* and *AffectsList*. The last of these lists must be generated, while the model provides the first two. As mentioned above, the *EnablingStateVariables* are the state variables that affect the enabling status of an action. Likewise, the *AffectedStateVariables* are the state variables whose value may change as a result of an action firing.

```
$\forall action \in \{Actions\}$
    $\forall stateVariable \in action.EnablingStateVariables$
        $if\,(action \notin StateVariableList[stateVariable].EnablingList)$
            $StateVariableList[stateVariable].EnablingList =$
                    $StateVariableList[stateVariable].EnablingList \cup \{action\}$
    $\forall stateVariable \in action.AffectedStateVariables$
        $if\,(action \notin StateVariableList[stateVariable].AffectedList)$
            $StateVariableList[stateVariable].AffectedList =$
                    $StateVariableList[stateVariable].AffectedList \cup \{action\}$
$\forall stateVeriable \in StateVariableList$
    $\forall action \in StateVariableList[stateVariable].AffectedList$
        $\forall action' \in StateVariableList[stateVariable].EnablingList$
            $if\,(action' \notin action.AffectsList)$
                $action.AffectsList = action.AffectsList \cup \{action'\}$
```

Figure 3.3: Connectivity List Generation Algorithm

A corner case, which this algorithm does not account for, is the case of an action enabled by a constant function. This might occur, for instance, in a SAN that has an input gate with a constant predicate enabling an activity. To handle this occurrence, we ensure that each action contains itself in its own affected list. Since the enabling function of an action is formalism-dependent, it cannot be determined when an enabling predicate is constant. Thus, there is a possibility that this process will generate an overly large connected list; however, it is more desirable to check enabling functions too often than too rarely. Typically, an action within a model will always affect itself when fired. Thus, this additional connection rarely causes significant changes within the final connectivity list.

Figure 3.4 shows a simple example of the connectivity list generation process using a SAN model. The input gate connected to the `A1` activity may enable the activity based either on a constant function or on a function dependent on the state of the model. The exact details of the function are not accessible to the simulator, since they are formalism-specific and cannot be generalized by evaluation during initialization. The action-to-state-variable lists provided by the model are shown to the right of the model. Using the algorithm in Figure 3.3 and adding the additional `A1` connection to the list results in the completed connectivity list shown in the lower right of Figure 3.4.



Action-to-State Variable Connectivity from the model

A1		A2		A3	
enabling	affected	enabling	affected	enabling	affected
	P1	P2	P2	P1	P1
			P3	P3	P3
					P4

Derived State Variable-to-Action Connectivity

P1		P2		P3		P4	
enabling	affected	enabling	affected	enabling	affected	enabling	affected
A3	A1	A2	A2	A3	A2		A3
	A3				A3		

Leading to Action-to-Action lists

A1 → A3

A2 → A2, A3

A3 → A3

Ensuring each action affects itself, the final lists are created

A1 → A1, A3

A2 → A2, A3

A3 → A3

Figure 3.4: Sample Connectivity List Generation

The final stage of initialization for the simulator engine is to fork off a child process for network communication. A separate process is used for this task both to facilitate asynchronous communication with the simulation manager and to prevent the parent simulation process from consuming processing time monitoring the network. Since there is only a small amount of communication required between the parent and child processes, POSIX (Portable Operating System Interface) signals are used for interprocess communications. There are essentially only two commands that the child process needs to send the parent: *finish* and *quit*. The finish signal is used to indicate that the simulation should report the current set of results to the manager and then terminate. This command uses the `SIGINT` signal. The *quit* signal indicates that the simulation is to terminate immediately. This instruction uses the `SIGQUIT` signal. By utilizing signal handlers to catch these commands, the parent process is able to use the POSIX `siglongjmp` function to "jump" out of the execution loop and "jump" back in at a different location. The return value from this function call varies depending on the signal received by the process. This allows the parent process to terminate the simulation "gracefully" when the simulation is completed. When running in simulation debug mode, the response to the `SIGINT` signal is slightly different. After each action fires, the simulator waits for a signal from the user before continuing execution of the next action. This allows the user to "step" through the simulation and observe the interactions of the model as it executes. Future implementations of the simulator may use this type of functionality to graphically step through the model execution.

For communication with the Java simulation manager, the child process uses standard BSD (Berkley Software Distribution) sockets. The simulation client is passed the listening port of the manager as a command line argument. After model instantiation, the child process initiates a registration transaction with the simulation manager. This registration process allows the simulation manager to identify the port the client uses to receive commands and verifies that communication is possible with the manager. The manager acknowledges the registration by sending a data packet to the client process. If any of these steps fail during initialization of the child, the client exits. This acts as a catchall mechanism to prevent the simulation engine from consuming processor time when it cannot report the results. This type of checking continues during the simulation, as the client will exit if it is not able to connect to the manager at any time during the simulation. In addition to the two commands sent to the parent process, the child also accepts a "ping" command from the manager, which it responds to directly. This command may be initiated by the user to ensure that the client is executing. A response to this command simply indicates that the client process exists and is executing the model. This is useful for the user in confirming that a simulation process has not exited or otherwise terminated during long simulations.

3.1.2. Data structures

Because the state changes of the model and accumulation of the reward variables for the model are accomplished by the model itself, the simulator engine has relatively few data structures of its own. Other than the future events lists and distribution class, which are discussed later in this chapter, the primary data structures within the simulator engine include stacks to manage reactivation events, arrays of pointers for race resampling events, a series of structures and unions to convert doubles into the proper network byte ordering, and arrays to hold batch observations. Singly linked lists, which are used for stacks in the simulator, and arrays are extremely simple programming elements. All linked list structures in the simulator allocate the maximum number of elements they will need when they are first instantiated. These elements are stored on a free list until they are required, and they are returned to the free list when completed. Since the simulator by nature is fairly small, this is an acceptable space-time trade-off as it prevents the time-consuming task of creating and destroying objects from slowing the simulation.

The simulator also uses a series of structures and unions to convert the batch observations, which are stored as doubles, into a standard network format. This conversion permits the simulator to run on a variety of computer architectures and allows distributed simulations to communicate using standard network formats. This network standard is referred to as *Big Endian* format. It simply specifies that as the bytes are serialized for network transport, they are placed in the bit stream with the most significant bit first. An architecture that works on data in *Little Endian* format (least significant bit first), such as the x86 architecture, must convert the data bytes to Big Endian before placing them into the network stream. The standard C and C++ libraries provide functions that facilitate this conversion for integer data types; however, they do not provide functions for floating point types. To overcome this problem, the simulator uses a structure composed of two long data elements, and forms a union of this structure and a double. By creating two of these unions, the double can easily be converted using the available C libraries. The double to be converted is stored into the first union. This allows the longs in the union to represent the first four and last four bytes composing the double. We can accomplish the byte swap by exchanging the longs and performing the standard C `htonl` function before storing the value in the second union. Figure 3.5 shows an example of the conversion process.

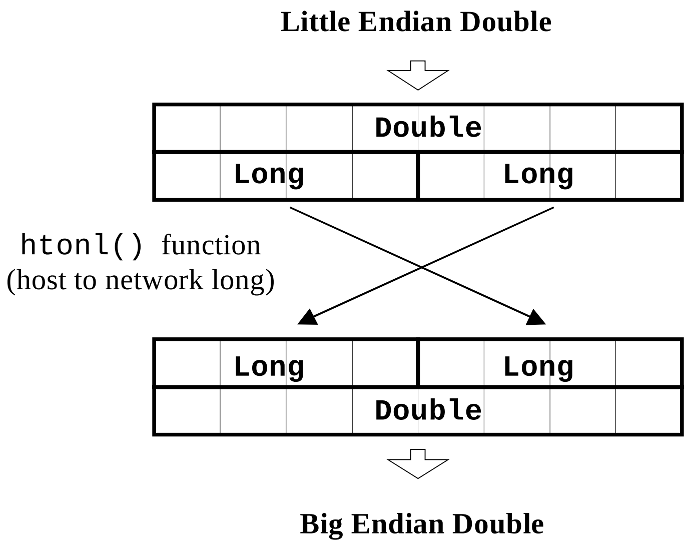


Figure 3.5: Conversion of Little Endian Double to Big Endian Double

The final data structure used in the simulation engine is the structure used to transmit the batch observation to the simulation manager. To store the observations as they are accumulated, a simple two-dimensional double array is required. The array contains as many columns as there are reward variables defined in the model, and enough rows to hold the observations between report intervals, with an additional row added to keep track of the number of observations recorded. In addition to this information, the simulation manager also requires the identification number of the simulation client and the experiment number to which the received observations apply. The simulation engine therefore creates a structure containing two doubles, necessary for identification of the client and the experiment, and the two-dimensional observation array. To simplify the socket implementation, the simulator uses a union of the described structure and a character array equal to the total size of the structure.

3.1.3. State change mechanisms

In order to maintain the extensibility of the Möbius environment, the simulator is not able to implement state changes directly within a model. Therefore, the models themselves must be able to correctly maintain their own state throughout the execution of the model. This allows the simulation engine to function more as a supervisor to the state change mechanisms rather than as the actual implementation of them. To initiate a state change within the model, the simulator uses the formalism-generic, `FireAction` method of the base model class. Each formalism must implement this method to correctly modify the state of the model. The simulator then functions as a scheduler and manager for the models, ensuring that actions are fired in the correct order and that reward observations are accumulated and recorded at the proper intervals.

3.1.4. Execution policy support

As discussed previously, the simulator is designed to support a wide range of execution policies, including race enabling memory, race age memory, race resampling memory, reactivation, preselection, and postselection. Pre- and postselection policies are implemented through specially designed actions called *groups*. As mentioned previously, these groups are essentially a set of actions related by their execution policy, which abstracts the particular action in the group that is to be executed. Although several actions within a group may be enabled at any one time, the group is only represented by one event on the future events list, and only one member of the group fires when the group is executed. Specifically, for a preselection group, the element of the group to be fired is determined at enabling time, when the event representing the group is placed on the future events list. Postselection groups select the element of the group that executes when the fire method of the group is called. This abstract notion of groups disrupts the behavior of the connectivity list optimization since it masks the true action that executes from the simulator. Thus, to allow groups to function efficiently in the simulator, the `FireAction` method returns the selected action from the group. This allows the simulator to check the enabling of only those actions that are affected by the selected action. Further discussion of groups can be found in [12].

Race memory policies specify how actions within a model behave when they become disabled or are required to resample their distributions. Race enabled actions are the standard execution policy for most formalisms. As discussed, this policy simply states that when an action becomes disabled, the representation of the event on the future events list is removed, and no record of the event is maintained. No special implementation is required for this policy, since the simulator is only required to remove the event from the events list if an action becomes disabled before the execution time is reached.

Race age actions are similar to race enabled; however, when the event is removed from the future events list, a record is stored of the portion of execution time the action "completed." When the race age action becomes reenabled, the portion of the event completed previously is factored in to the time sample before the new event is to be executed. The implementation of this policy is done by storing a double on each action representing the fraction of the previous event execution time that was completed. When the event becomes reenabled, this value is factored into the new time sample of the distribution by multiplication of the fraction previously completed. In order to allow actions to become enabled and disabled multiple times before finally reaching completion, the following equation is used to calculate the fraction of the execution time completed:

$$FractionComplete = \frac{CurrentTime - StartTime}{ExecutionTime - StartTime}(1 - FractionComplete) + FractionComplete,$$

where *StartTime* is the time when the action was enabled for the current event, *ExecutionTime* is the time the action was to execute for the current event, and *CurrentTime* is the current virtual simulation time. This algorithm represents an approximation of the conditional distribution required to accurately analyze the effect of age on the execution time of the action. Future simulator implementations or revisions may utilize the conditional distribution calculations to match more accurately the analytical analysis of this policy [14].

Race resampling actions are required to sample their distribution function after each action in the model fires. Therefore, the only time that a race resampling memory action will fire is if it is the next action within the entire model to fire. This property allows for a great simplification in the treatment of these actions. Since the action will only fire if its event time is earlier than any other event time on the future events list, there is no reason to place it on the event list unless it will fire next. Thus, by maintaining a list of these actions, and sampling their distribution after each action fires, it can be determined if any resampling actions are to fire. If the conditions are not achieved, only the action that is to fire next is placed on the future events list. This optimization prevents a great deal of unnecessary insertions and removals from the event list, thus speeding the simulation. An important note in the implementation of this policy is that placing resampling actions in the connectivity lists generated by the simulator results in redundant enabling checks on the resampling actions. This is because all resampling actions are checked at each state change in addition to the actions indicated by the connectivity lists. To prevent this extraneous checking, resampling actions are excluded from the connectivity lists during list generation.

The reactivation policy is the most complex of the execution policies implemented. As stated earlier, reactivation actions have both a reactivation predicate and function. If the reactivation predicate is true when the action becomes enabled, the action is said to be reactivatable and a flag on the action is set. If at any time before firing, the reactivation function of a reactivatable action becomes true, the action is to be removed from the events list, resampled, and, if still enabled, placed back on the events list. Implementation of this policy requires that the base action class contain two reactivation methods, one for the predicate and the other for the reactivation function. Also required is a flag that may be set by the simulator to identify an action as having a true reactivation predicate at enabling time.

A corner case for the reactivation function is an action that becomes enabled with both a true predicate and function. These actions are allowed to be placed on the future events list; however, they must be checked after the firing of the next action to determine whether the reactivation function is still true. The affects list associated with each action cannot account for this occurrence, because there is no guaranteed correlation between the action that fires and the reactivation action in question. To handle this problem, the simulator makes a stack of reactivation actions that meet this criteria as they are placed onto the events list. After the firing of the next event, they are popped off the stack, and their reactivation conditions are checked. If the reactivation function is still true, they are removed from the events list; otherwise, they remain on the list. Although this occurrence is usually rare, the simulator must allocate a stack large enough such that all reactivation actions could be placed onto the stack if the proper conditions are found. This requires the simulator to know the total number of actions in the model that may become reactivated at some point during model execution. Since predicates and functions may be state-dependent, the model formalisms must set the reactivatable flag on each action to "true" for initialization. This allows the simulator to account for all actions that may become reactivatable at some point during model simulation. The reactivation flag is reset to "false" after this initialization stage is complete and maintained entirely by the simulator through model execution.

Except in the above-mentioned scenario, reactivation actions are handled in much the same way as other actions within the model. The only additions are the storage of a memory of the predicate at enabling time and the checking of the reactivation function at the same time as the enabling function when the predicate flag is set. Formalisms must be careful to append any state variables referenced in the reactivation function onto the enabling list of the action such that the proper connectivity list can be generated. This step ensures that the firing of an action that may affect the return value of the reactivation function causes the simulator to check the return value of the reactivation function.

Race age, resampling, and reactivation execution policy support in the simulator is enabled through preprocessor commands during compilation. This acts as an optimization to prevent time-consuming policy checks from being executed when actions in the model do not require them. The default execution policy is race enabling memory. If a model is created that requires the age, resample or reactivation functionality, the Java classes of the models must pass the simulator `AGE`, `RESAMPLE`, or `REACTIVATION` as preprocessor compile options.

3.1.5. Reward accumulation interface

Typically, formalism-specific simulators are also responsible for collection of reward variables while the model executes. Unfortunately, this produces a highly integrated and formalism-specific simulation and reward variable accumulation package. In order to maintain consistency with the Möbius object-oriented paradigm, the simulator and reward variable accumulation are completely separate. This allows the simulator to be easily extended or even replaced entirely by development of new simulation methods. The simulator accesses the solvable model using a series of methods and data elements. Discussions of the specific interfaces are split between terminating and steady state simulation because each of these modes of operation for the simulator interacts slightly differently with the reward variable interface.

Terminating simulation

Reward variable accumulation during terminating simulation is a much simpler interface than steady state because it requires fewer interactions by the simulator. A *replication* for a terminating simulation is defined as the time period from the beginning of the simulation to the time at which the last reward variable finishes accumulation. Since a reward variable may only accumulate one observation per replication, the simulator can wait until the end of the replication to record the observation that is reported to the manager. Thus, the simulator is able to record the values of the observation for each reward variable at the end of the replication. At the beginning of each new replication, the simulator initializes the solvable model using the `BeginAccumulate` method on the model. The simulator is provided information about when the first variable to begin accumulation starts and when the last variable ends through access to data members on the solvable model. This allows the simulator to execute the model up to the period in which accumulation begins without using processor time to check the reward variables.

During the accumulation period, the simulator uses the `Accumulate` and `AccumulateAll` methods of the solvable model to distinguish between results updating within a variable accumulation period and updating at the end points. The `AccumulationStartTime` and `AccumulationStopTimes` are used to identify these different periods. The `AccumulateAll` method updates these times to the earliest start time and earliest stop time, after the current time, of the variables being collected. Each action within the model also contains a pointer to a list of reward variables that are affected by the action. This is similar to the affects list that the simulator generates to determine which actions are affected by the firing of other actions. When the reward variable affects list is not empty and the start and stop times do not indicate that the `AccumulateAll` method should be called, the `Accumulate` method is used to update the variables.

Communication between the simulator engine and reward variable model also occurs during the firing of each action within the model. This is done using the `FireAction` method on the reward model. By communicating via this method, the reward model is able to continuously update internal monitors of the reward variables.

Steady state simulation

The steady state interface provides a slightly more complicated interface for the simulator. The steady state accumulation periods are called *batches*, rather than replications as in the terminating simulation. The reason that this mode of operation is more complicated for simulation is that reward variables are not required to have the same batch length or initial transient times. Thus, different reward variables may accumulate at different rates during the simulation. The simulator must be aware of this and send the results to the manager when any variable reaches the maximum number of observations for a report. This is the reason that the simulator keeps an index of the number of results from each variable as part of the data array. The entire data array is sent to the manager regardless of whether it is completely full so that a consistent interface is provided to the manager.

For this mode of simulation, the simulator itself tracks the end of accumulation time for each reward variable. This is done using the `Stop` data member on each variable. This element indicates the simulation time when the variable has finished accumulation. The simulator scans through the variables and selects the minimum stop time of the variables. When this time is reached, the values for any observations completed are stored in the results array. The collected variables are then reset such that the next batch can begin to accumulate. This eliminates the methods required by terminating simulation to reset all variables at once. Interactions during the accumulation period of the rewards remain fairly consistent with the terminating simulation. The primary difference is that the `AccumulateAll` method is not required. This is because start and end of accumulation periods are explicitly handled by the collection algorithms.

3.1.6. Numerical distributions and random number generators

Besides the actual functionality of the simulator as a solver, the simulator library provides the Möbius environment with random number streams and random number distributions. These components are contained in the `Distribution` base class, which is extended to the `UserDistributions` class to allow new distribution functions and random number generators to be added. The base class itself offers two random number generators, one based on the Tausworthe [15] and the other based on the Lagged Fibonacci series [16]. Distributions based on these random number streams include beta, binomial, deterministic, Erlang, exponential, gamma, geometric, hyper-exponential, log normal, negative binomial, normal, triangular, uniform, and Weibull [15]. The distribution class is instantiated outside of any particular solver and is thus usable throughout the model. By referencing the `TheDistribution` pointer, both the distributions and random number streams are accessible. Implementation of the specific random number streams and distributions is identical to that found in *UltraSAN* [2]. Parameters for the distributions can be found in Appendix A.

3.1.7. Future events list management

The management of future events is one of the most time-consuming components of discrete event simulation. The reason for this is that actions may be placed on and removed from the events list many times before ever firing. If the insertion, removal, and sorting algorithms of the events list are not efficient, the speed of simulation will be greatly reduced. As an optimization to event removal from the events list, actions within the Möbius framework have built into them a pointer used to reference the event representing them on the events list. If the event should become disabled prior to the event executing, the event may be removed with an O(1) operation. Insertion and sorting of the events list are left to the specific implementation of the events list in use. As with all Möbius components, the future events list is designed to be extended. The present implementation is a linked list with support for probabilistically selecting events with identical completion times. This class is derived from the base `EventList` and base `Event` classes. These base classes define simple methods that the simulator uses to interact with the events list. These methods include operations such as insertion into the list, removal from the list, returning the next event to fire, removing all elements from the list, and printing the list, for debugging purposes.

As mentioned, the present implementation of the simulator future events list uses a linked list. The constructor of this class creates a free list, which contains enough elements that every action in the model may be represented on the events list if necessary. This prevents the need for constructing and destroying event list elements during run time. This is another space-time trade-off for the simulator, favoring a faster executable that may use slightly extra memory during execution. The events list is also required to support selection of an event from a group of events with the identical execution times. This is done by evaluating the `Rank` and `Weight` methods on each action. The event is to be selected from those actions with the highest rank, using the weight of each action to bias the selection. The current implementation accomplishes this task by using a quadruply linked list. The two primary links on the list function as a standard doubly linked list where events with different execution times are stored in ascending execution-time order. When an event that has the same execution time as another event is added to the list, the new event is added to the secondary set of links on the event associated with the specific execution time. The basic structure of the list is shown in Figure 3.6. When the "multiple event group" reaches the head of the event list, the removal process scans through the secondary list for the actions with the highest rank. At the same time, the weights of these actions are summed. After this initial pass, a random number is selected using the random number generator and scaled by the sum of the weights. A final pass is then made through the group event list to determine which event is selected. This capability greatly enhances the simulator's ability to execute models containing multiple deterministic actions.

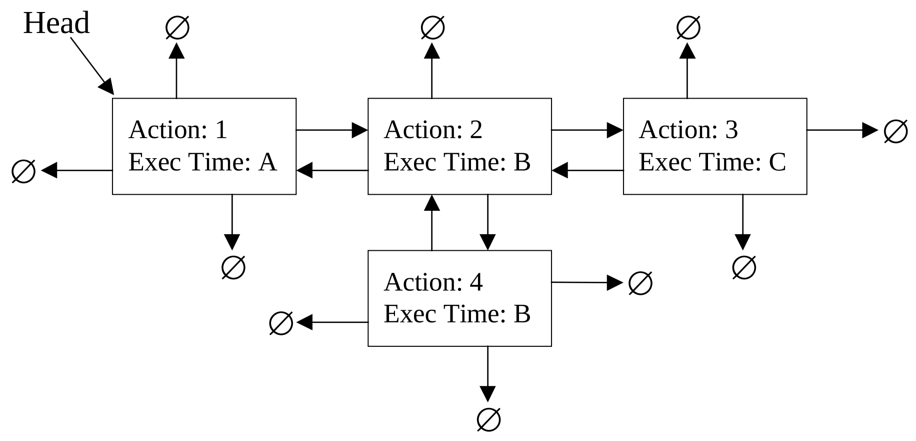


Figure 3.6: Linked List Future Events List Structure

3.1.8. Socket communication

The simulation engine and Java manager use TCP/IP sockets for communication of batch observations, client registration, and client commands. Although the stream TCP protocol incurs slightly more overheads than a datagram protocol such as UDP, the reliability of the transport makes it a worthwhile choice. The child process, within the simulation engine, uses the POSIX `select` function to effectively sleep while waiting for instructions from the manager. This prevents the child process from unnecessarily occupying the processor while the parent simulation is executing. For simplicity of design, only double data types are used for network communication.

Reporting of batch observations to the manager is done by the parent simulation engine. This eliminates a significant amount of interprocess communication that would otherwise be necessary between the parent and child processes. However, this division of work does delay the simulator from creating more observations while network communication proceeds. Fortunately, most operating systems provide nearly instantaneous network functions for the small data packets used by the simulator. This is because of buffering at the lower layers of the transport protocol. Thus, we only use an extremely small percentage of the total processor time for network communications.

3.2. Simulation Management, Reward Collection, and User Interface

Simulation management by the Möbius simulator is achieved entirely through Java classes. Java provides a flexible and powerful platform for object-oriented, multi-threaded, asynchronous applications. This allows for extremely efficient handling of multiple simulation clients communicating via TCP/IP sockets, continuous reward variable updates, and a "user friendly" graphical interface. The architecture of the Java components of the simulator can be broken down into three main components: management of simulation clients, data collection, and the user interface. Each of these is discussed separately in the following sections.

3.2.1. Client management

Client management in the simulation interface is handled by a series of threads and objects. Threads are used to allow components of the simulator to branch off and execute independently. These act much like a "light-weight" child process. The benefit of the thread is that it simplifies asynchronous communication and processing. Threads used by the manager spend most of their time waiting for interruptions either by timers expiring, processes terminating, or receiving data from the client. Thus the threads spend very little of their running time consuming processor cycles. This is important, because the simulation manager is designed to be run in parallel on the same system as the simulation client without adversely affecting the client's performance.

For each processor selected to run a distributed simulation, a thread is created to manage the processor. A nondistributed simulation is treated the same, except that there is only one processor allocated. The purpose of this thread is to find an available experiment and start the simulation client running on the experiment. When the simulation client finishes, the thread finds the next experiment and restarts the client. The class responsible for this thread is the `ProcessorManager`. This class interacts almost exclusively with the `ExperimentObject`, which represents a single experiment on the model and provides methods for limiting access to the experiment and maintains a thread for calculating statistical results on reward variables. Both the managers and the experiment objects are maintained as vectors of objects. Vectors provide a built-in, linked-list-like structure to the Java language. The manager traverses through the vector of experiments until an experiment is found that is available for execution. Available experiments are those that have not already been completed and those that do not already have the maximum number of processes executing the experiment. When an experiment is found, the manager registers the processor with the experiment, ensures that the thread used to calculate reward variables for the experiment is instantiated, and starts a simulation client to begin execution of the experiment.

The `ProcessorManager` also handles the task of sending control commands to the client simulation engine. This is done by a series of methods that create a socket and connect to the child process of the client. If a response is required by the client, such as with a “ping” request, the thread creates a server socket and waits for a response. The thread also creates and controls a debugging window when a trace level three simulation is selected. This is handled by opening the input stream from the client process and copying the data to the window. To step through a simulation, the manager sends a command to the client to break it from its suspended state. The client then executes one action firing before returning to the suspending state.

3.2.2. Data collection

Data collection by the simulation manager is also handled by a group of threads. The controlling thread in this group is the `NetworkThread`. Upon instantiation, this thread creates subthreads to read data from the network and transfer the data to local buffers. The network thread creates one of these threads for every processor manager available. This helps to eliminate any possibility of the client simulation stalling while the manager allocates a new reader to accept the data. When a connect signal is received from a client simulation, the Java accept method returns a socket for communicating with the client. This socket is passed to the `NetworkReader` subthread. The network reader first copies the data from the client into a local buffer. This is done as a simple byte read to avoid excessive synchronization calls through the Java streams. Once the raw data is stored locally, the network connection is closed and the data is formatted and placed into a double storage array for use by the experiments.

The network portion of the simulation manager makes extensive use of free lists to reuse objects and prevent unnecessary processor usage instantiating new objects when they are required. Objects reused include the object-arrays to store the formatted data from the network, the object-arrays to store the raw data from the network, and the network reader threads themselves. The formatted data objects are created as needed by the reader threads if none are available. This might occur if the length of time between calculation of results is significantly longer than the time interval between reports from the clients. When a free formatted data object is available, the reader thread extracts it from the free list vector, copies the new data into the array, and places it on a vector of new data to be included with an experiment. A thread used to calculate the reward variables returns the object to the free list vector after the data is accounted for. Similarly, the network reader threads are reused by the network thread. The network thread removes a reader from the free list vector of those available; if one is not available, it is created. The socket for the reader thread is set and the thread resumes execution. After receiving the data from the network and placing the formatted data object on the correct new data list of the experiment, the reader thread places itself back onto the list of free readers. Figure 3.7 graphically shows the interactions of these components as the manager executes.

As mentioned, a separate thread is used to calculate the reward variables as the model is executed. This thread is associated with the experiment object. Thus, there is one calculation thread for each experiment to be executed. The calculation thread manages a group of calculation objects. These objects are derived from a base object to handle specific types of reward variable calculations. Derived types include mean, variance, interval estimators, and distribution estimators. Other types may easily be derived as needed. The managing thread passes to each calculation object the vector of new data objects as they become available. The specific calculation objects either update the current values of their measurement on the fly or store the data into a private array for evaluation at the end of the simulation. The latter option is required by the variance calculations so that the confidence interval of the variance may be calculated. After each calculation object has been given an opportunity to use the data, the objects are returned to the free list, where they may be extracted and removed repeatedly.

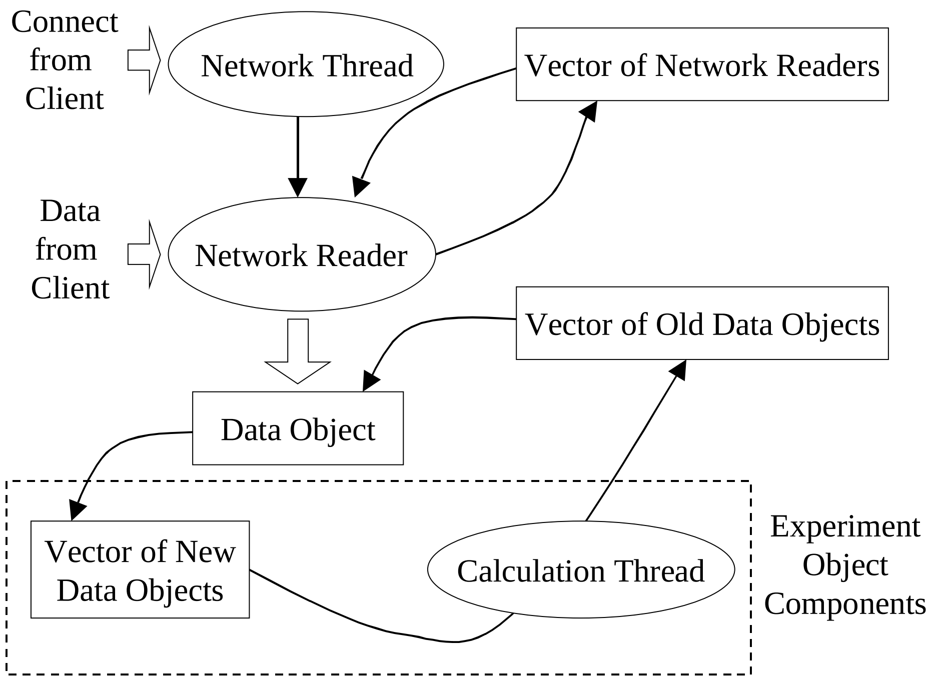
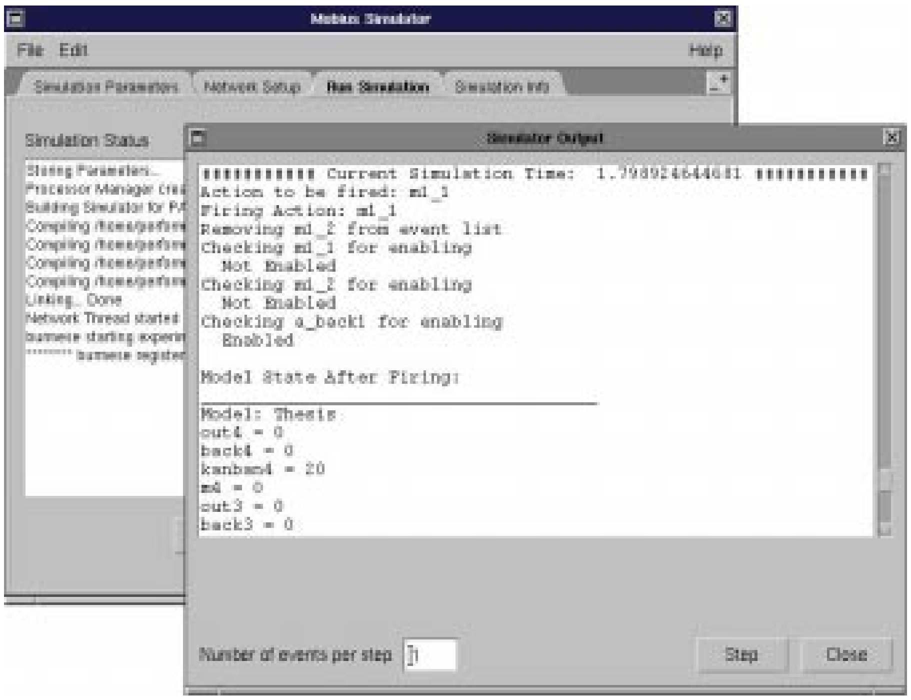

Connect from Client
Network Thread
Vector of Network Readers
Data from Client
Network Reader
Vector of Old Data Objects
Data Object
Vector of New Data Objects
Calculation Thread
Experiment Object Components

Figure 3.7: Data Flow through the Simulation Manager

To prevent the Java manager from dominating the use of the processor during the simulation, only the mean calculation object is updated on the fly. The other objects simply record the values for later calculations or update counters without calculating confidence intervals. This is the reason that the mean estimation is the only variable displayed on the interface window as the simulation is running. When an experiment is completed, the calculation thread allows each calculation object to perform final calculations on the observations recorded. The method used for these final calculation returns a results object containing the information about the final value of the variable. These observations are collected and grouped into a list, which may later be used by a results analysis package.

3.2.3. User interface

The Java graphical user interface to the Möbius simulator provides an assortment of useful debugging options, real-time results observations, and client information not available in other, comparable simulators. The simulator is able to provide a great deal of information about the specific interactions of a simulated model when a trace level 2 or 3 simulation is run. The trace level 2 simulation output is intended to be written to a file for review outside of the Möbius package. Trace level 3 simulation actually allows the user to step through the execution of each action within a model. During this process the simulation client pauses execution until the user advances the simulation. This is accomplished using a series of sleep functions, which are interrupted using commands sent to the simulation engine by the user. An example of the simulator interface for this output is shown in Figure 3.8. These trace levels are activated in the simulator by passing certain preprocessor options to the makefile as it is written. The extra information supplied is thus only compiled into the simulation executable if the trace level is desired. This allows the simulator to provide a significant amount of information without slowing the simulation engine during normal execution.



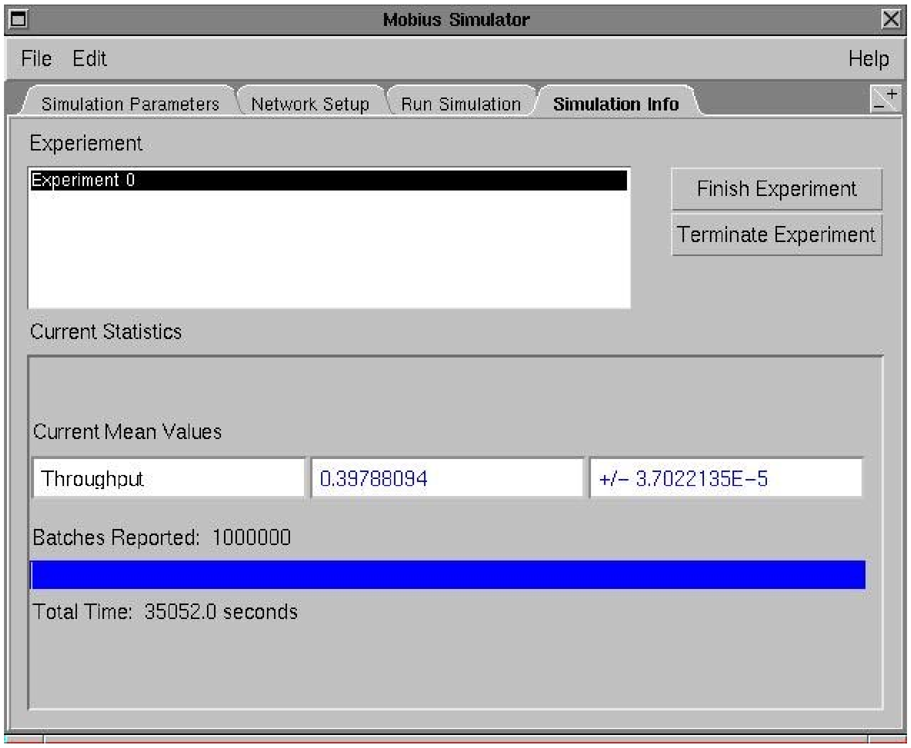
```
Storing Parameters...
Processor Manager cre
Building Simulator for P
Compiling /home/perform
Compiling /home/perform
Compiling /home/perform
Compiling /home/perform
Linking... Done
Network Thread started
burmese starting experi
********* burmese registe
```

```
########### Current Simulation Time:  1.798924644681 ###########
Action to be fired: m1_1
Firing Action: m1_1
Removing m1_2 from event list
Checking m1_1 for enabling
  Not Enabled
Checking m1_2 for enabling
  Not Enabled
Checking a_back1 for enabling
  Enabled

Model State After Firing:
______________________________
Model: Thesis
out4 = 0
back4 = 0
kanban4 = 20
m4 = 0
out3 = 0
back3 = 0
```

Figure 3.8: Simulator Model Debugging Output

Real-time observations are provided to the user as the model is executed using components contained in the calculation thread. When an experiment is selected, so that the progress and results are shown, the panel displayed is actually contained within the mean calculation object. The values shown within this panel are updated as the mean object receives new data and updates the values. Additional information is available to the user through the ability to query a specific processor to determine which experiment it is executing and when the experiment was started. An example of this information is shown in Figure 3.9. This ability to query the clients is helpful in tracking the progress of the simulator through the complete execution of the model.



Experiement

Experiment 0

Current Statistics

Current Mean Values

Throughput	0.39788094	+/- 3.7022135E-5

Batches Reported: 1000000

Total Time: 35052.0 seconds

Figure 3.9: Simulation Progress Window

Complete details of the graphical interface and user-configurable options are discussed in Appendix B.

4. RESULTS AND COMPARISONS

For analysis of the Möbius simulation engine, we compare the performance results to those of the terminating and steady-state simulation engines in *UltraSAN*. As the model used for these comparisons is exclusively for analysis of the solver, we do not focus on the results of the simulations. All simulations are executed on HP C180 workstations using 180 MHz PA-RISC 2.0 processors. Each workstation is equipped with 1 MB of data cache and 1 MB of instruction cache. System memory ranges from 128 to 512 MB of RAM. The systems have insignificant additional loads during simulations. Network communications is via 100 Mbps switched Ethernet. Except where noted, each simulation was run for exactly 10 000 batches each of 1000 time units with a 100 time unit initial transient. One impulse reward variable is defined on the `out4` activity of the model, and the lagged Fibonacci random number generator was used for the Möbius simulations.

The model used for these comparisons was previously used by Ciardo and Tilgner [17] 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 and moves to another stage or returns to be worked on again. Objects leaving stage 1 may enter either stage 2 or stage 3. Objects are recombined in the final stage, after processing from stage 2 or 3. For those familiar with SANs, an example of the model is shown in Figure 4.1. The capacity of each stage 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. A complete description of the model parameters can be found in [17].

We begin the investigation of the simulator performance on this model by varying the number of tokens in the Kanban places between 1 and 10. This is done using a global variable within the model. Since there are four instances of these places within the model this also varies the total number of tokens in the model between 4 and 40 in steps of 4 tokens. The effect of increasing the number of tokens in the model is to increase the activity within the model. More actions within the model are enabled concurrently as a result of the increased available tokens. This greatly increases the time the simulator must spend checking actions for enabling and the time necessary to maintain the future events list. The results of this test for steady state and terminating simulation are shown in Figure 4.2 and Figure 4.3.

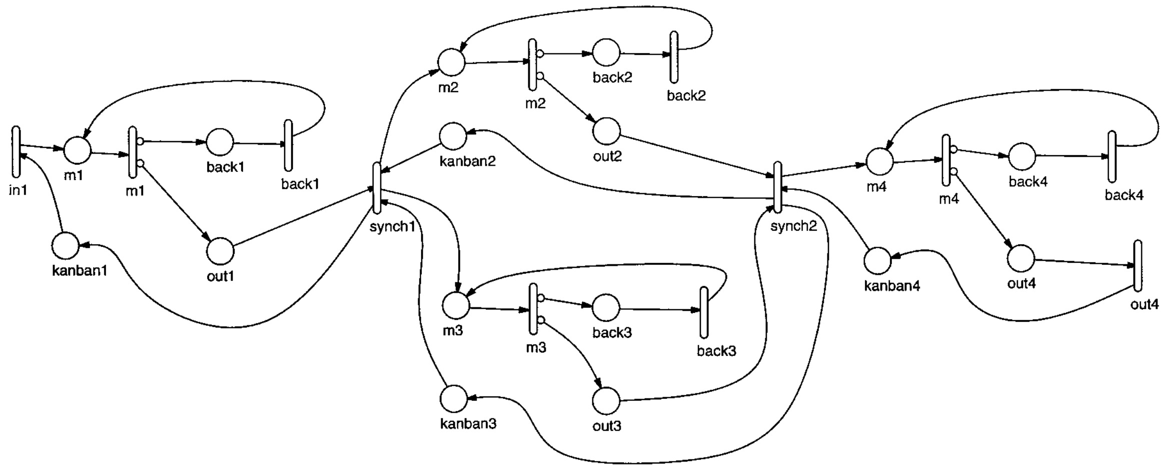


Figure 4.1: The Kanban Test Model

As seen by these comparisons, the Möbius simulator provides a significant performance increase over the *UltraSAN* simulation engines for both steady-state and terminating simulations. The average performance increase over *UltraSAN* for this model is 63.4% for steady state simulation and 53.6% for terminating simulation. Terminating simulation has a slightly reduced performance over steady state due to the reward variable interface and model state resets prior to each replication. For these tests, the steady state Möbius simulation engine achieves an average of over 150 000 state changes per second. The *UltraSAN* steady state simulation engine averages almost 89 000 state changes per second.

To test the performance of the distributed simulation interface, we again use the Kanban model and perform a steady state simulation analysis, using one to four processors. The execution times for these simulations are shown in Figure 4.4. The overall execution times for each simulation are marginally slower than those shown in Figure 4.2, because of

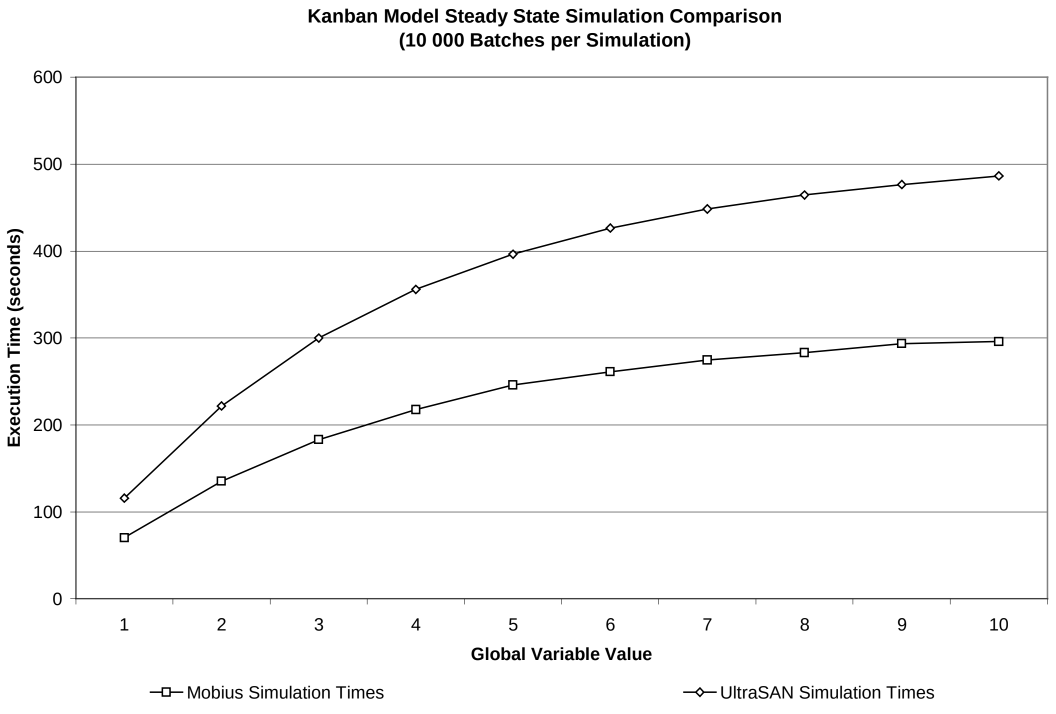


Figure 4.2: Steady State Simulation Performance by Varying Initial Model State

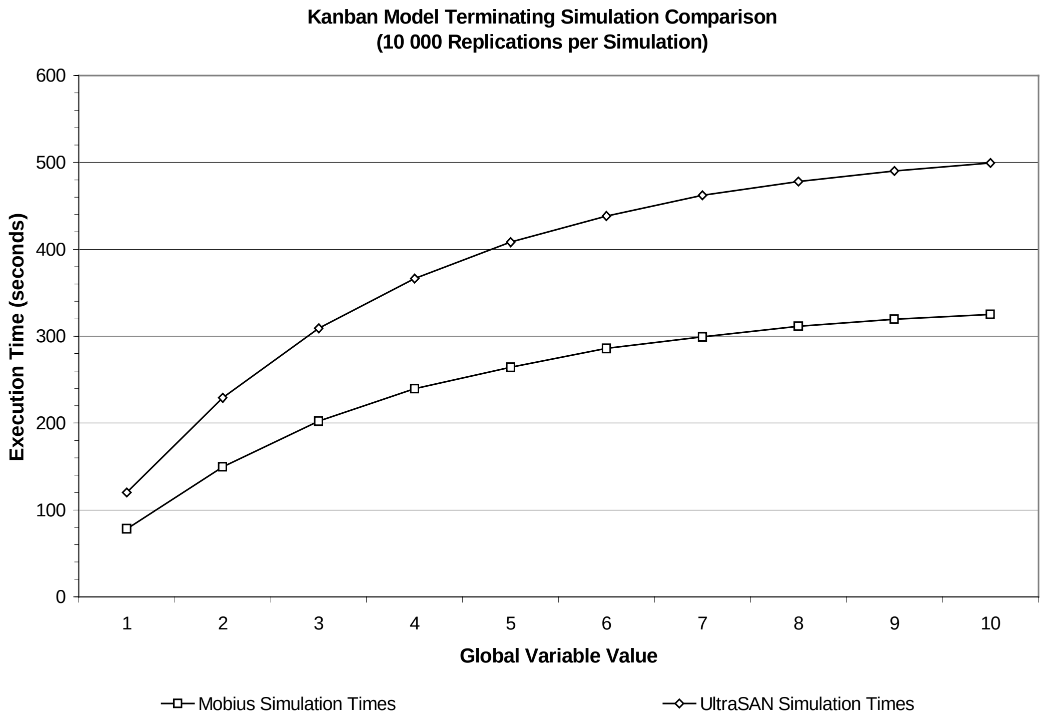


Figure 4.3: Terminating Simulation Performance by Varying Initial Model State

nonoptimized compile-time options. However, as is quite apparent from this example, the Möbius simulation manager scales exceptionally well to distributed simulations. User-definable parameters, such as batch size, do impact the overall performance of the simulation. This is seen in the two "3 Processor" graphs shown in Figure 4.4. By defining a batch size of one thousand, we configure the simulation engines to report blocks of observations containing one thousand observations each. Since among three processors this does not divide evenly into the total of ten thousand batches there is a portion of processing time wasted. Redefining the batch size to 1112 observations per report (estimating the optimal $1111.\overline{1}$) substantially improves the multiprocessor speed increase. Figure 4.5 shows the increased performance of the distributed Möbius simulation engine compared with an ideal linear increase. Variations from linear are likely to be associated primarily with network overhead and deviations in processor time consumption of the systems.

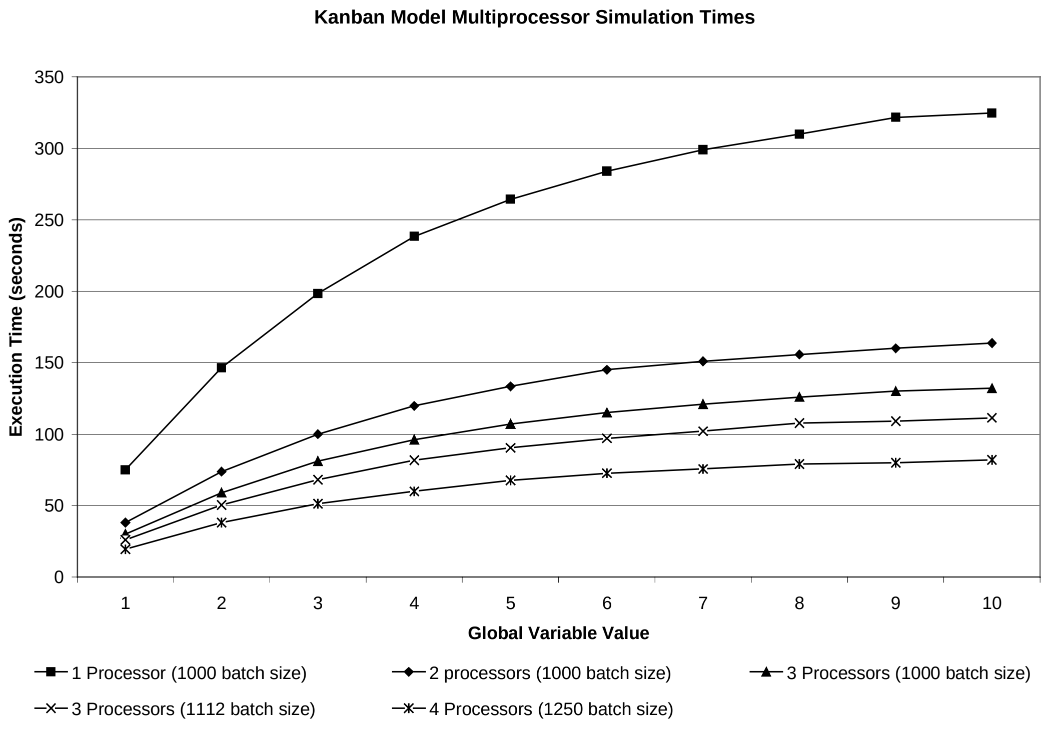


Figure 4.4: Möbius Distributed Simulation Performance

To test the scalability of the simulator, we used the composed model editor to define a variable number of replications of the Kanban model. Each replication of the model runs independently; however, the total number of model components managed by the simulator increases linearly with the number of model replications. No components between the replicated models were shared for these tests, and each simulator ran 10 batches for each model configuration. Figure 4.6 shows the effect of the increased model size on the number of state changes executed per second by the simulation engine.

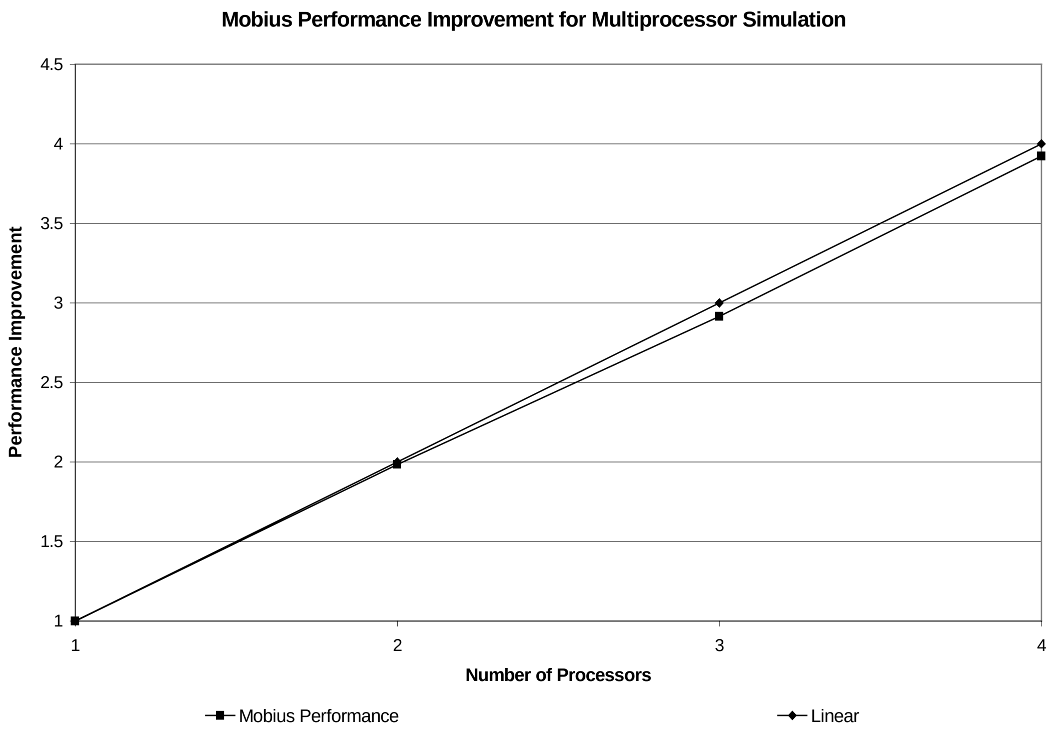


Figure 4.5: Distributed Simulation Performance Scaling

We see from the example that both *UltraSAN* and Möbius have significantly reduced performance for extremely large models. For Möbius, this is due primarily to the overhead of object-orientation and the increase in time consumed managing the future events list. The object overhead for Möbius is a side effect of the formalism-independent structure of the model. Since each component is derived from a set of core components, the structure of the model cannot be represented with a set of simple data elements, as the specific-formalism simulator is able to do. Also, as the model increases in size, the total number of state changes per simulation batch increases. For instance, the 250-replication Kanban model requires approximately 6.6 million state changes for a 1000 time unit simulation. This is a significant increase from the 660 000 state changes for the 25-replication simulation. As the number of state changes increases, the management of the future events list begins to dominate the processing time consumed by the model. However, in spite of these overheads, the Möbius simulator continues to perform better than the *UltraSAN* simulator.

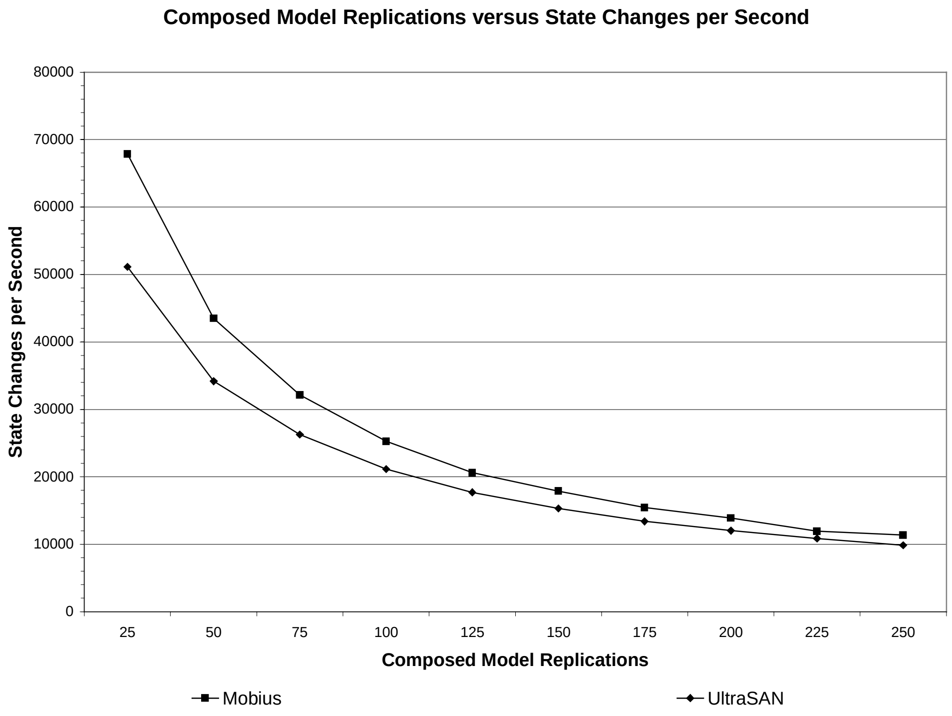


Figure 4.6: Simulator Scalability for Composed Models

As seen through these examples, the increased generality of the Möbius environment does not necessarily decrease the performance of the solution methods. By using insightful and innovative data structures we are able to produce a full-featured, formalism-independent simulation engine that is able to surpass the performance of some formalism-specific simulators.

5. CONCLUSION AND FUTURE RESEARCH

The main objective of this work was to develop a discrete event simulation package capable of steady state, terminating, and distributed simulation for use in the Möbius modeling environment. This objective was achieved by

1. Creating a simulator that accesses models using only a core set of functions to provide a formalism-independent interface.
2. Developing data structures to efficiently manage model components for optimal model execution.
3. Designing support for an assortment of execution policies to allow modelers increased flexibility in their models.
4. Developing a graphical interface that simplifies the use of the simulator and provides run-time model statistics.
5. Optimizing the performance characteristics of the simulator to exceed those of comparable formalism-specific simulation packages.

The utility of this new simulation package is shown in the examples demonstrated in the previous chapter. In each case, the Möbius simulator achieved significant performance increases over the *UltraSAN* simulator for identical models. As the Möbius modeling package grows and matures, new formalisms will be developed and added, allowing the true flexibility and extensibility of the simulator to be shown.

Although the simulator in this thesis is mature and ready for use on industrial applications, areas for improvement and future research remain. The simulator developed in this project is designed to allow components to be easily extended or replaced as the field of discrete event simulation advances and as system modelers desire new features. This allows for simple additions and expansions to the tool. For instance, in the simulation engine, the linked-list data structure used for future events list management is effective for small to medium models. However, advanced structures may provide a substantial performance increase for large models. Other areas for research and additions into the simulation engine include new random number generators, numerical distributions, and execution policy support.

The Java simulation manager also provides an excellent platform for future research. The manager itself is designed to be independent of the simulation engine. Thus, by developing new simulation engines that communicate using the protocols developed in this project, the features of the manager may continue to be used. Extensions to the manager, such as new statistical calculation methods, are also easily included. This may be done simply by extending the existing classes. By maintaining the extensibility of the simulation package throughout the design and implementation, we have created an excellent platform for future research and implementations of simulation solution methods.

APPENDIX A: DISTRIBUTION PARAMETERS

The Möbius simulator provides a number of continuous and discrete numeric distributions for use in model analysis. These distributions may be accessed using the `TheDistribution` pointer global to the solvers. The available distributions and their parameters are described below.

- Beta(α_1, α_2):

  Density: $$f(x) = \begin{cases} \dfrac{x^{\alpha_1 - 1}(1-x)^{\alpha_2 - 1}}{\mathrm{B}(\alpha_1, \alpha_2)} & \text{if } 0 < x < 1 \\ 0 & \text{otherwise} \end{cases}$$

  where $\mathrm{B}(\alpha_1, \alpha_2) = \int_0^1 t^{z_1 - 1}(1-t)^{z_2 - 1}\,dt$ for any real number $z_1 > 0$ and $z_2 > 0$.

  Mean: $\dfrac{\alpha_1}{\alpha_1 + \alpha_2}$

  Variance: $\dfrac{\alpha_1 \alpha_2}{(\alpha_1 + \alpha_2)^2 (\alpha_1 + \alpha_2 + 1)}$

- Binomial(t, p):

  Distribution: $$F(x) = \begin{cases} 0 & \text{if } x < 0 \\ \sum_{i=0}^{\lfloor x \rfloor} \binom{t}{i} p^i (1-p)^{t-i} & \text{if } 0 \le x \le t \\ 1 & \text{if } t < x \end{cases}$$

  Mean: tp

  Variance: $tp(1-p)$

- Deterministic$(value)$:

  Mean: $value$

  Variance: 0

- Erlang (m,β):

  See Gamma (α,β), for integer values of ($m = \alpha$) > 0.

- Exponential$(rate = \lambda)$:

  Density: $$f(x) = \begin{cases} \lambda e^{-\lambda x} & \text{if } x \ge 0 \\ 0 & \text{otherwise} \end{cases}$$

  Mean: $\dfrac{1}{\lambda}$

  Variance: $\dfrac{1}{\lambda^2}$

- Gamma (α,β):

Density: $f(x) = \begin{cases} \dfrac{\beta^{-\alpha} x^{\alpha-1} e^{-x/\beta}}{\Gamma(\alpha)} & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$

where $\Gamma(z) = \int_0^\infty t^{z-1} e^{-1} dt$ for any real number $z > 0$

Mean: $\alpha\beta$

Variance: $\alpha\beta^2$

- Geometric (p):

Distribution: $F(x) = \begin{cases} 1 - (1-p)^{\lfloor x \rfloor + 1} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$

Mean: $\dfrac{1-p}{p}$

Variance: $\dfrac{1-p}{p^2}$

- Hyperexponenital $(rate_1 = \lambda_1, rate_2 = \lambda_2, p)$:

Distribution: $F(x) = \begin{cases} p\left(1 - e^{-\lambda_1 x}\right) + (1-p)\left(1 - e^{-\lambda_2 x}\right) & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$

Mean: $\dfrac{p}{\lambda_1} + \dfrac{(1-p)}{\lambda_2}$

Variance: $2\left(\dfrac{p}{\lambda_1^2} + \dfrac{(1-p)}{\lambda_2^2}\right) - \left(\dfrac{p}{\lambda_1} + \dfrac{(1-p)}{\lambda_2}\right)^2$

- Lognormal (μ, α^2):

Density: $f(x) = \begin{cases} \dfrac{1}{x\sqrt{2\pi\alpha^2}} e^{\frac{-(\ln x - \mu)^2}{2\alpha^2}} & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$

Mean: $e^{\mu + \alpha^2/2}$

Variance: $e^{2\mu + \alpha^2}\left(e^{\alpha^2} - 1\right)$

- Negative binomial (s, p):

Distribution: $F(x) = \begin{cases} \sum_{i=0}^{\lfloor x \rfloor} \dbinom{s+i-1}{i} p^s (1-p)^i & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$

Mean: $\dfrac{s(1-p)}{p}$

Variance: $\dfrac{s(1-p)}{p^2}$

- Normal $(mean = \mu, variance = \sigma^2)$:

Density: $f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/2\sigma^2}$ for all real numbers x

Mean: μ

Variance: σ^2

- Uniform $(lowerbound = a, upperbound = b)$:

Density: $f(x) = \begin{cases} \frac{1}{b-a} & \text{if } a \le x \le b \\ 0 & \text{otherwise} \end{cases}$

Mean: $\frac{a+b}{2}$

Variance: $\frac{(b-a)^2}{12}$

- Triangular (a, b, c):

Density: $f(x) = \begin{cases} \frac{2(x-a)}{(b-a)(c-a)} & \text{if } a \le x \le c \\ \frac{2(b-x)}{(b-a)(b-c)} & \text{if } c < x \le b \\ 0 & \text{otherwise} \end{cases}$

Mean: $\frac{a+b+c}{3}$

Variance: $\frac{a^2 + b^2 + c^2 - ab - ac - bc}{18}$

- Weibull (α, β):

Density: $f(x) = \begin{cases} \alpha\beta^{-\alpha} x^{\alpha-1} e^{-(x/\beta)^\alpha} & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$

Mean: $\frac{\beta}{\alpha}\Gamma\left(\frac{1}{\alpha}\right)$

Variance: $\frac{\beta^2}{\alpha}\left\{2\Gamma\left(\frac{2}{\alpha}\right) - \frac{1}{\alpha}\left[\Gamma\left(\frac{1}{\alpha}\right)\right]^2\right\}$

All variables may be specified as doubles. Further descriptions of the distributions may be found in [15].

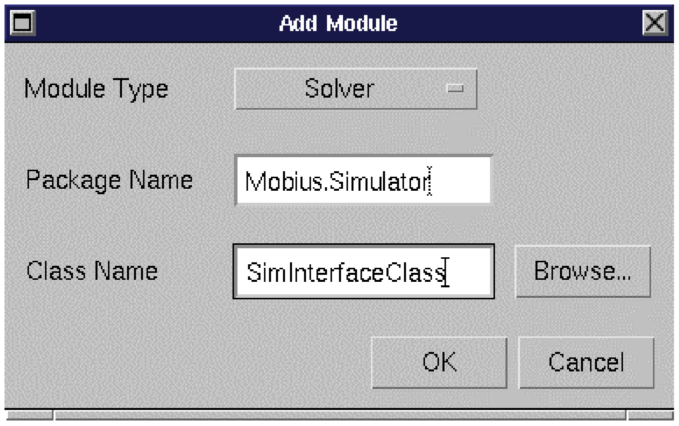
APPENDIX B: SIMULATOR USERS MANUAL

B.1. Möbius Simulation

The Möbius modeling package includes both a terminating and steady state simulation engine. These simulators generate confidence intervals for the variables specified in the reward model using the batch means method for steady state and the replication method for terminating simulation. Both simulators use the same interface and are selectable using a checkbox on the initial simulator dialog. Once installed, the simulator may be selected from the "Solver" menu on the Möbius control panel. When a new simulator 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 several tabs. Each of these tabs represents a stage in the setup and solution of the simulation.

B.2. Installing the Möbius Simulator Module

To install the Möbius simulator into the Möbius control panel, select the "Add Module" menu item from the "Modules" menu on the control panel. Fill in the entries as shown in Figure B.1.



Module Type: Solver
Package Name: Mobius.Simulator
Class Name: SimInterfaceClass

Figure B.1: Adding the Simulator Module

In order to use the simulator, 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 may either be set 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 switches are also essential to allow the simulator to operate reliably. It is recommended that the maximum native stack size and maximum Java heap size be specified. The following startup line is recommended:

```
java –ss1M –mx64M StartMobius
```

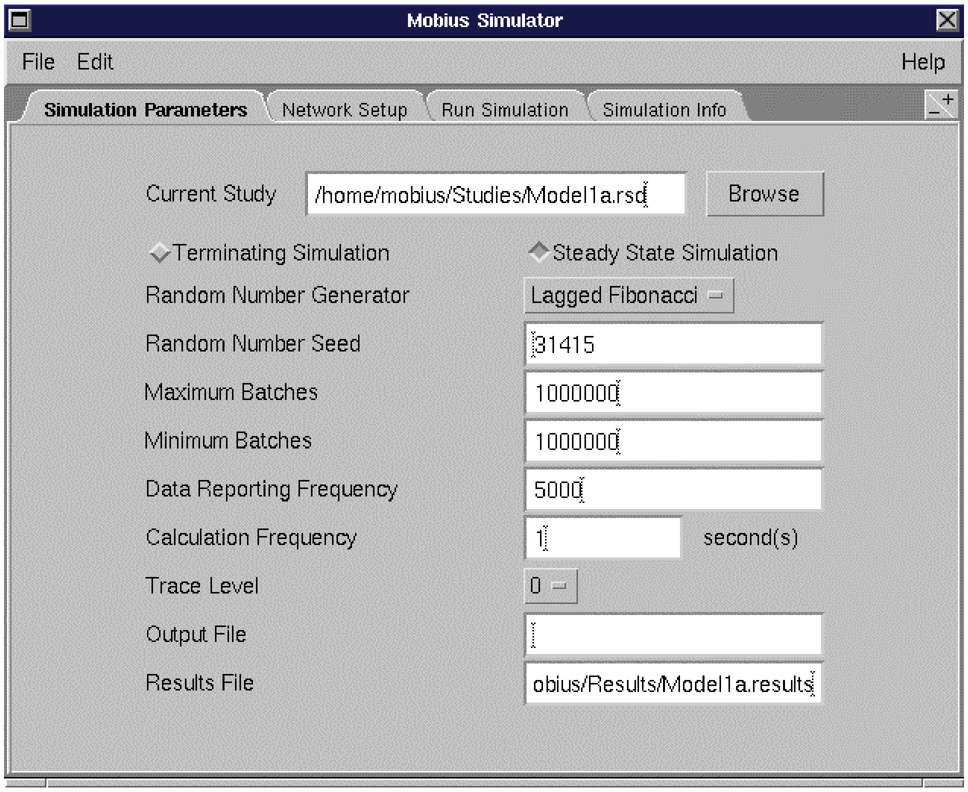
The maximum heap size option (mx) may be adjusted larger, but caution is advised in allocating a larger native stack size (ss).

B.3. Simulation Parameters

When the simulator is started from the File→New→Solver→Mobius Simulator menu item on the control panel, a dialog is presented to specify the study for the simulation. Specify the saved file name for either a range or set study. Once a valid study is provided, the panel shown in Figure B.2 is presented. This panel allows the specific parameters of the simulation to be configured.

Simulation parameters include:

- **Current Study:** The location of the saved study file for this simulation run.
- **Terminating/Steady State Simulation:** The simulation type based on the measures desired from the model.
- **Random Number Generator:** Lagged Fibonacci or Tausworthe random number generators. The lagged Fibonacci, by Knuth [18], provides an extremely numerically stable generator; however, it is slightly slower than the less stable Tausworthe generator.
- **Random Number Seed:** The seed used for the Java random number generator. This generator produces the corresponding seeds for each processor in the simulation.
- **Maximum Batches:** The maximum number of batches or replications to execute per experiment.



Current Study	/home/mobius/Studies/Model1a.rsd
Terminating Simulation	Steady State Simulation
Random Number Generator	Lagged Fibonacci
Random Number Seed	31415
Maximum Batches	1000000
Minimum Batches	1000000
Data Reporting Frequency	5000
Calculation Frequency	1 second(s)
Trace Level	0
Output File	
Results File	obius/Results/Model1a.results

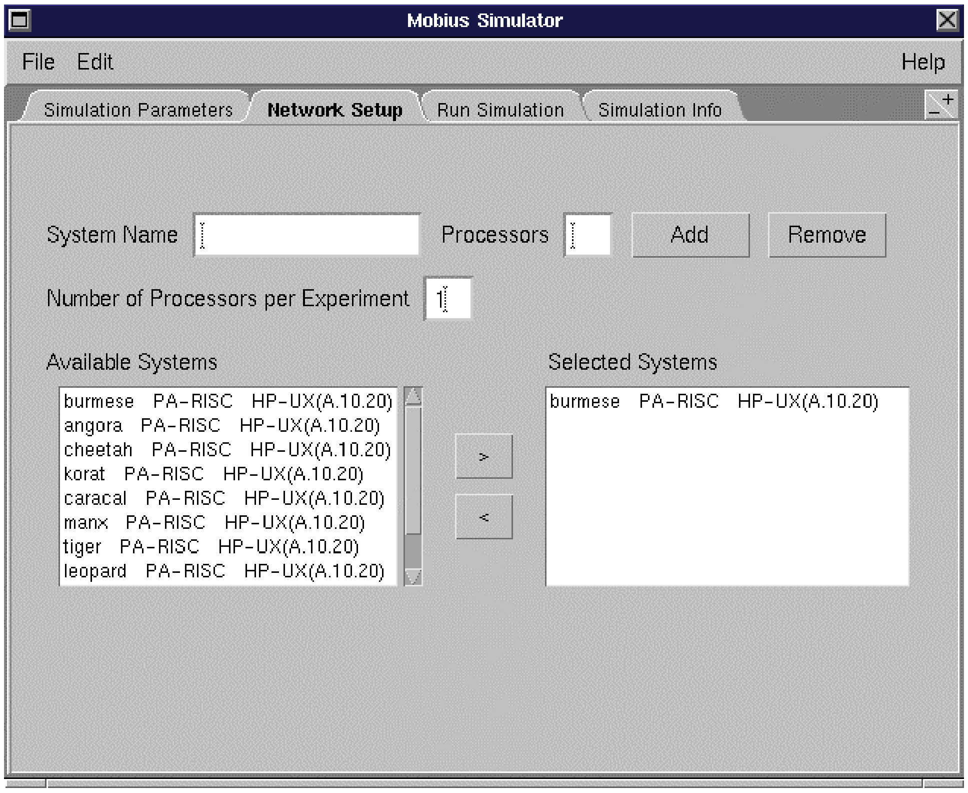
Figure B.2: Simulation Parameters Panel

- **Minimum Batches:** The minimum number of batches or replications to execute per experiment.
- **Data Reporting Frequency:** The number of observations a client simulation engine should generate before reporting results to the manager.
- **Calculation Frequency:** The amount of time the calculation thread sleeps before checking for new data and updating observation statistics.
- **Trace Level:** The level of output from the simulator. Values are:
    0. No trace information
    1. Information on the generation of connectivity list and execution policy specific data structures.
    2. Detailed output of every state of the model and event firing. Also include the elements of the future events list (must also specify an output file)
    3. Same as level 2, except the simulation may be stepped through, firing one action at a time and observing the model changes.

- **Output File:** The file in which to store the output from the simulation. If none is specified, output is to system `/dev/null`.
- **Results File:** The file in which to store the results of the simulation. If none is specified, the user is prompted for a file upon completion of the simulation.

B.4. Network Setup

The second tab on the simulator interface configures the network setup of the simulator. This is where processors to run simulations are configured. Figure B.3 shows this interface. Computer systems may be added to the available list by entering the system name in the text field, specifying the number of processors the computer has in the next field, and clicking the "Add" button. The interface will then query the system to determine the architecture and operating system. To remove a system from the list of available systems, highlight the system in the list box and click the "Remove" button. The list of



Number of Processors per Experiment: 1

Available Systems

```
burmese   PA-RISC   HP-UX(A.10.20)
angora   PA-RISC   HP-UX(A.10.20)
cheetah   PA-RISC   HP-UX(A.10.20)
korat   PA-RISC   HP-UX(A.10.20)
caracal   PA-RISC   HP-UX(A.10.20)
manx   PA-RISC   HP-UX(A.10.20)
tiger   PA-RISC   HP-UX(A.10.20)
leopard   PA-RISC   HP-UX(A.10.20)
```

Selected Systems

```
burmese   PA-RISC   HP-UX(A.10.20)
```

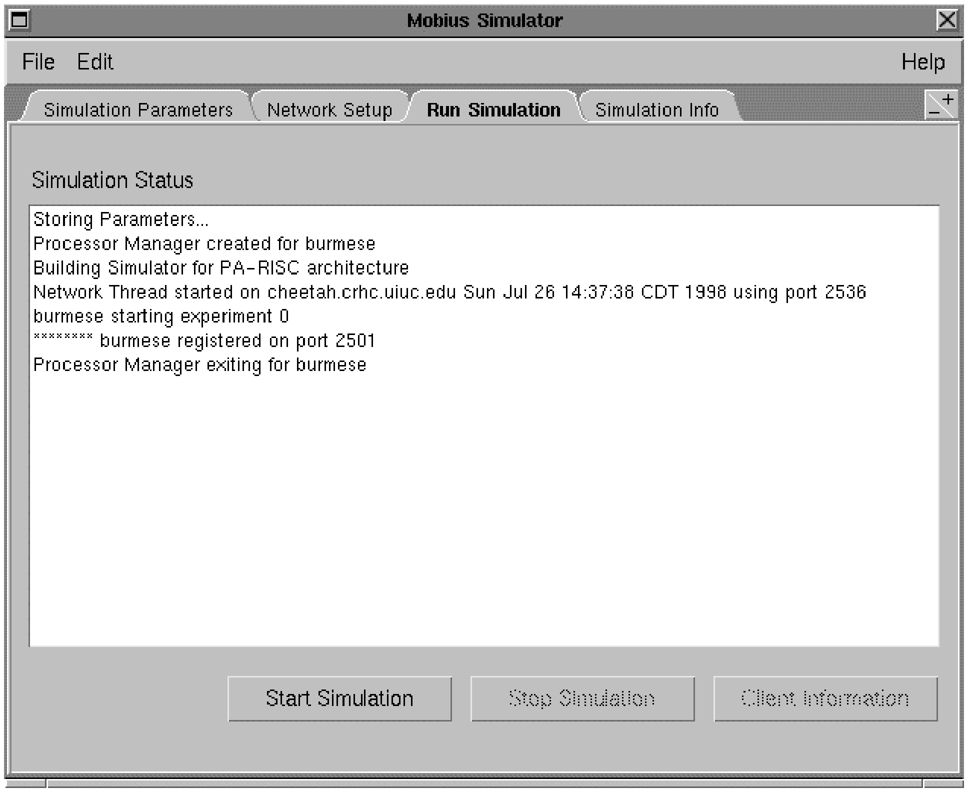
Figure B.3: Network Setup Panel

available systems is stored in the user's `~/.mobius` in the file `sim.cp`. This file may only be edited using the simulator interface.

To select processors to run the current simulation, highlight the processor in the "Available Systems" list and select the move right arrow (">"). The system should then appear in the "Selected Systems" list. To reverse the process, use the move left arrow ("<"). To configure the simulator to allow more than one processor to simulate a single experiment concurrently, modify the "Number of Processors per Experiment" field accordingly.

B.5. Run Simulation

The "Run Simulation" tab, shown in Figure B.4, allows the user to start and stop the simulation, as well as to query specific processors during the simulation. When the simulation is started, the interface validates all the entries that the user has selected to ensure proper operation. Any errors are displayed. The user is then asked to save the simulation to a file. The location and name of this file will provide a basis for the executable compiled for the



Simulation Status

```
Storing Parameters...
Processor Manager created for burmese
Building Simulator for PA-RISC architecture
Network Thread started on cheetah.crhc.uiuc.edu Sun Jul 26 14:37:38 CDT 1998 using port 2536
burmese starting experiment 0
******** burmese registered on port 2501
Processor Manager exiting for burmese
```

Figure B.4: Run Simulation Panel

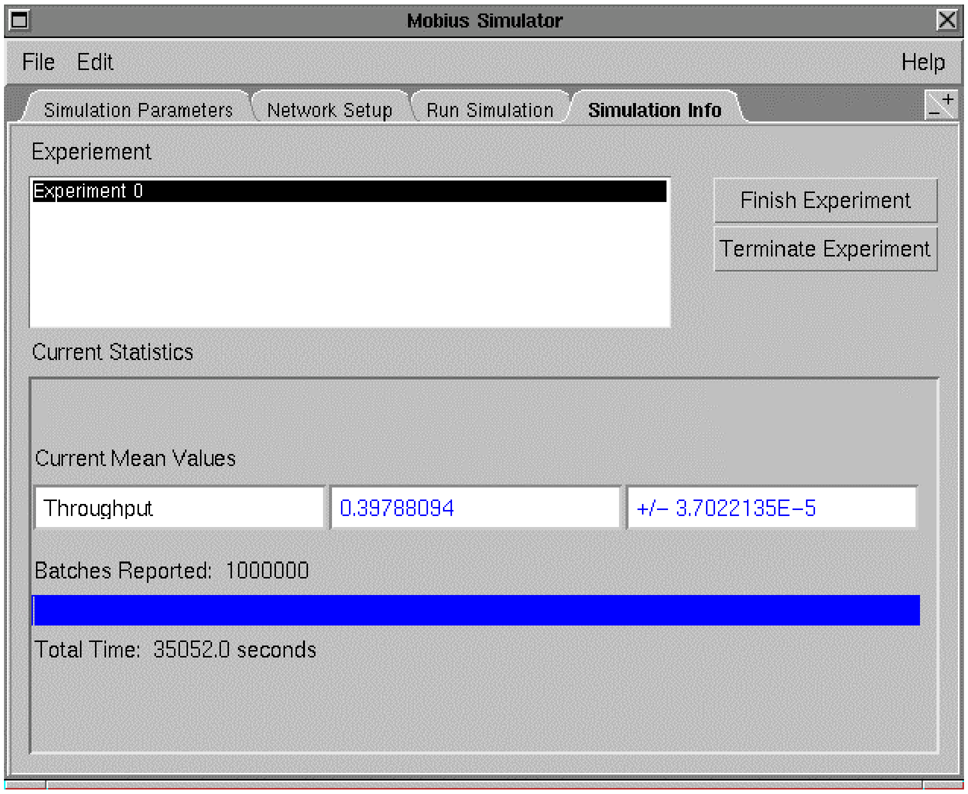
simulation. Thus, models may be developed on private file systems and run distributed as long as the simulator is saved in a shared file system. However, the system that compiles the executable must also have access to the model itself and thus must be able to access the private directories. This is easily accomplished by ensuring that the first processor of each architecture, selected in the network setup, has access to the private locations.

After validation and saving, the simulator compiles the simulator for each architecture of the processors selected. As the simulation is run, information about the progress of the separate systems is displayed in the status window. Selecting the “Stop Simulation” button causes the simulation to terminate immediately without storing results. The “Client Information” button launches a dialog that allows the user to query information from a specific processor.

B.6. Simulation Information

The “Simulation Info” tab of the simulator, shown in Figure B.5, provides on-the-fly progress indicators for the experiments as they are running. To view the current values of the variables, select an experiment from the list of experiments in the upper left window. If the experiment selected has not yet been started, the current statistics window will display a “no results” message. Only the current values of the *mean* statistics are displayed, as these are the only ones calculated during the simulation.

If the user determines that the values of the reward variables have become acceptable before the simulation completes, the user may complete the experiment using either the finish or terminate buttons. The “Finish Experiment” button sends a signal to the simulator to report the observations already recorded and exit. The “Terminate Experiment” button performs the same function, except that any results currently recorded by the simulator but not reported are lost. The difference between these two finishing methods is most noticeable for long simulations, in which the time intervals between reports may be significant.



Experiement

Experiment 0

Current Statistics

Current Mean Values

Throughput 0.39788094 +/- 3.7022135E-5

Batches Reported: 1000000

Total Time: 35052.0 seconds

Figure B.5: Simulation Information Panel

B.7. Simulator Tips and Tricks

The simulator interface is quite amenable to the novice user; however, some advice is necessary to achieve optimal functionality. The user-configurable option that is most important in making the simulator function correctly is the “Data Reporting Frequency” field. The simulation clients collect blocks of observations to report to the manager, because of the excessive communication necessary to report only the observations from each batch or replication. For small models, the C++ simulation engines can easily outrun the Java interface in system output. Thus, it is crucial that the user not specify a value that is too small for simple models. For larger models, which require minutes or more to complete a batch or replication, smaller values may be appropriate.

The simulator is also optimized to avoid compiling the lower components of a model when it is unnecessary to do so. During normal Möbius use, this should work sufficiently. However, when changes are made to the model code manually, it is often necessary to recompile the executable without restarting the simulator. To do this, simply resave the study to which the simulator is associated and restart the simulation. The simulator interface keeps a record of the time stamp of the study file and recompiles the entire model when the time stamp changes.

REFERENCES

[1] John Meyer, “On evaluating the performability of degradable computing systems,” *IEEE Transaction on Computers*, vol. C-22, pp. 720-731, August 1980.

[2] *UltraSAN User’s Manual Version 3.0*, The University of Illinois, 1995.

[3] F. Bause, P. Buchholz, and P. Kemper, “QPN-Tool: For the specification and analysis of hierarchically combined queueing Petri nets,” in F. Bause, H. Beilner, Eds., *Quantitative Evaluation of Computing and Communication Systems*, *Lecture Notes in Computer Science*, no. 977. Berlin: Springer-Verlag, 1995, pp. 224-238.

[4] K. K. Goswami, R. K. Iyer, L. Young, “DEPEND: A simulation-based environment for system level dependability analysis,” *IEEE Transaction on Computers*, vol.46, no.1, pp. 60-74, January 1997.

[5] C. Lindemann, “DSPNexpress: A software package for the efficient solution of deterministic and stochastic Petri nets,” in *Proceedings of 6th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, Edinburgh, Great Britain, pp. 15-29, 1992.

[6] B. P. Zeigler, *Object-Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems*. New York: Academic Press, 1990.

[7] 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.

[8] M. Ajmone Marsan, G. Balbo, A. Bobbio, G. Chiola, G. Conte, and A. Cumani, “The effect of execution policies on the semantics and analysis of stochastic Petri nets,” *IEEE Transactions on Software Engineering*, vol. 15, no. 7, pp. 832-846, July 1989.

[9] W. H. Sanders, “Construction and solution of performability models based on stochastic activity networks,” Ph.D. dissertation, The University of Michigan, Ann Arbor, Michigan, 1988.

[10] C. H. Sauer and E. A. MacNair, *Simulation of Computer Communication Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1983, pp. 33-53.

[11] F. Bause, “Queueing Petri nets: a formalism for the combined qualitative and quantitative analysis of systems,” In *5th International Workshop on Petri Nets and Performance Models*, Toulouse, France, October 1993, pp. 14-23.

[12] J. M. Doyle, “Development of the Möbius abstract model specification,” M.S. thesis, The University of Illinois, Urbana, IL, in progress.

[13] J. Sowder, “State space generation techniques in the Möbius modeling framework,” M.S. thesis, The University of Illinois, Urbana, IL, 1998.

[14] D. D. Deavours, “The Möbius framework,” manuscript in preparation.

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

[16] D. E. Knuth, *The Art of Computer Programming*, Vol. 2, *Seminumerical Algorithms*. Menlo Park, CA: Addison-Wesley Publishing Company, 1981, pp. 25-31.

[17] 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.

[18] D. E. Knuth, “RANARRAY: Portable random number generator recommended in Seminumerical Algorithms, 3rd edition,” July 1997, http://www-cs-faculty.Stanford.EDU/~knuth/programs/rng-double.c.