# MODEL COMPOSITION WITHIN THE MÖBIUS MODELING FRAMEWORK

BY

AARON JAMES STILLMAN

B.S., University of Illinois at Urbana-Champaign, 1997

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1999

Urbana, Illinois

# ABSTRACT

Complex systems are made of large numbers of components, where each component may itself be composed of multiple subsystems. The size and complexity of such systems makes it impossible to design and study them using ad hoc methods. Thus, there is a growing need for powerful modeling environments to assist in the design and maintenance of complex systems. The Möbius modeling framework addresses this need by providing an object-oriented, extensible framework that allows rapid integration of custom formalism editors for model construction, simulation, and analytical/numerical solution. Each formalism editor provides the user with a specific language for expressing models.

The Möbius framework supports multiple modeling formalisms, methods for model composition and connection, and a way to integrate multiple analytical/numerical- and simulation-based model solution methods. An abstract functional interface specifies a set of methods that each modeling formalism in the framework must implement, and that can be used by models expressed in multiple modeling formalisms to interact with each other. The Möbius framework currently supports three different model language types: atomic models, composed models, and solvable models. In a typical design flow, the user will (1) define self-contained representations of system components using atomic models, (2) connect some combination of atomic models to form composed models, (3) create a solvable model by defining measures of interest on the composed model using reward variables, and (4) obtain numerical results for the system by using a solvable model.

This thesis focuses on the second step of the aforementioned design flow. Specifically, we have developed a generic state-sharing framework that allows a flexible means of model composition between any combination of atomic and composed models. We demonstrate the versatility of our state-sharing framework by implementing two composer formalisms within Möbius: the Replicate/Join formalism, which combines models into tree-like structures, and the graph formalism, which allows arbitrary connections between models. We believe that these two composer formalisms will serve as templates for the development of future composer formalisms.

# ACKNOWLEDGMENTS

First and foremost, I thank God for giving me peace and strength in all of my work. I also thank my parents for their continued prayers and support over the years. And a special thank you goes out to all of my friends who have always been there for me during the project, especially Brianna Sharp, Kevin Hsu, Tomomi Katsu, Kevin Koo, Jesse Mangler, Andrea Conway, and Judy Karlovsky.

I would like to thank my advisor, William H. Sanders, for helpful technical guidance along the way. Thanks to Dan Deavours, Patrick Webster, Dave Daly, and Jay Doyle for all of your help and for being great teammates on the Möbius project!

I would like to thank the Defense Advanced Research Projects Agency, Information Technology Office, for funding under contract DABT63-96-C-0069.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

The Möbius modeling framework was designed to support model specification across multiple modeling formalisms and to support the simulation and analytic solution of models. The driving motivation behind the framework is that a specific modeling formalism, such as stochastic activity networks [1, 2] or queueing networks, may be appropriate for the representation of a particular portion of a system being modeled, but not appropriate for an entire system. If a user is allowed to specify each part of the system in an appropriate representation, it becomes easier for the user to attain accurate results using simulation and analytical/numerical methods. The Möbius modeling framework [3] works toward this goal by allowing the designer great flexibility in model specification and analysis.

In order to explain the Möbius modeling framework, we first need to give a working definition of a model. A *model* is composed of "state" and "actions." The *state* of a system is some abstraction of the system's behavior and characteristics at some time. In Möbius, we divide the state of a system into distinct, non-overlapping *state variables*. The union of all the states of the system's state variables is the total state of the system.

*Actions* define the way the model's state may change. We say that an action *fires* when it induces the change of state associated with it. When fired, actions may change the state of one or more state variables. An action *affects* a state variable if it is capable of changing its state. In this thesis, we will explain how we can share state variables across models to build larger, more complex models.

A *formalism* is a language for expressing a model within the Möbius framework, and will often use only a subset of the options available within the framework. In this thesis, we will distinguish between two types of modeling formalisms: atomic and composed. An *atomic model* [4] is competely defined in one modeling formalism and is a self-contained representation of either part of a system or the entire system. On the other hand, a *composed model* is a collection of other models joined together by sharing states or actions. It should be noted that a composed model may be joined to other atomic or composed models to form a new composed model. In our design, atomic models are meant to be completely independent of composed models–they will record no information about which composed models include them. Likewise, a composed model formalism should not require any formalism-specific details of its constituent models. These two requirements assure that new atomic and composed model formalisms can be added to an implementation of the framework without changing existing formalisms.

Once the user finishes defining a model to the desired level of detail, he or she is then ready to analyze the model. This invariably will involve the user defining measures of interest so that specific parts of the system's behavior may be studied [5]. These measures of interest (also known as *performance* or *dependability variables*) may be thought of as observation points in the system, for which data will be collected during simulation or analytical/numerical solution.

Möbius makes use of two main techniques to provide the user with useful information regarding the defined measures of interest: analytical/numerical solution and Monte Carlo simulation. Analytical/numerical solution requires the determination of the complete continuous-time Markov chain (CTMC) for the system; to accomplish this, we use a state space generator. State space generation entails the determination of all of the unique, reachable states for the system. Details regarding state-space generation in Möbius can be found in Sowder [6]. Once the com-

plete CTMC is determined, we use mathematical solvers to determine either the transient or steady-state values for the measures of interest.

The second way that the Möbius user may obtain information regarding the defined measures of interest is through Monte Carlo simulation. Simulation does not require generation of the full state space; instead, random paths will be traced through the system's state space. As the paths are traced, observations will be made on the values of the performance variables. We use statistical techniques on the observed values to estimate the transient or steady-state values of the performance variables. The collected data will be observed until a result can be presented with the desired degree of confidence and to the desired degree of precision. Of course, both of these criteria are probablistic in nature. For more details regarding simulation in Möbius, refer to Williamson [7].

In this thesis, we will present methods of combining atomic models to form composed models within the Möbius modeling environment. First, we will introduce the state-sharing framework established by Jay M. Doyle and myself as a generic means of constructing composed models from atomic models. Then, we will present two composition formalisms that we have implemented in Möbius, namely, the Replicate/Join and graph formalisms, which use our state-sharing framework. These two composer formalism implementations demonstrate how new composer formalism interfaces can be built and integrated into the Möbius tool. Futhermore, they serve as prototypes that future composer formalism designers may use to create new composer formalism interfaces. This thesis will explain the design issues encountered in designing the two composer formalism implementations, and give examples of features that we have added to reduce the user effort required to design complex models.

# CHAPTER 2

# STATE-SHARING FRAMEWORK

## 2.1 Introduction

Fundamental to model composition within the Möbius framework is the notion of state sharing between different atomic models. In this chapter, we will discuss the mechanisms by which the Möbius user may share state between atomic models to create composed models. We will introduce the Möbius approach to state sharing, discuss implementation issues, and demonstrate how the user can use our state sharing framework to create models that have shared state.

We will refer to the sharing of state variables with state variables as *equivalence* sharing and the sharing of state variables with functions as *functional*, or *unidirectional*, sharing. When two state variables are equivalently shared, they essentially become the same state variable. This technique can be used as a means of connection between dissimilar models or as a convenient way of representing systems with inherent symmetry.

To demonstrate how equivalence sharing may be used to build a composed model, consider a superscalar processor that is capable of issuing instructions to any of several identical functional units from a common instruction queue. The superscalar processor can be viewed as the combination of several identical functional units, where each functional unit contains an instruction fetch and decode stage, instruction buffer, and execution core. Instead of having three separate functional units, we can reduce system cost by making the instruction fetch and

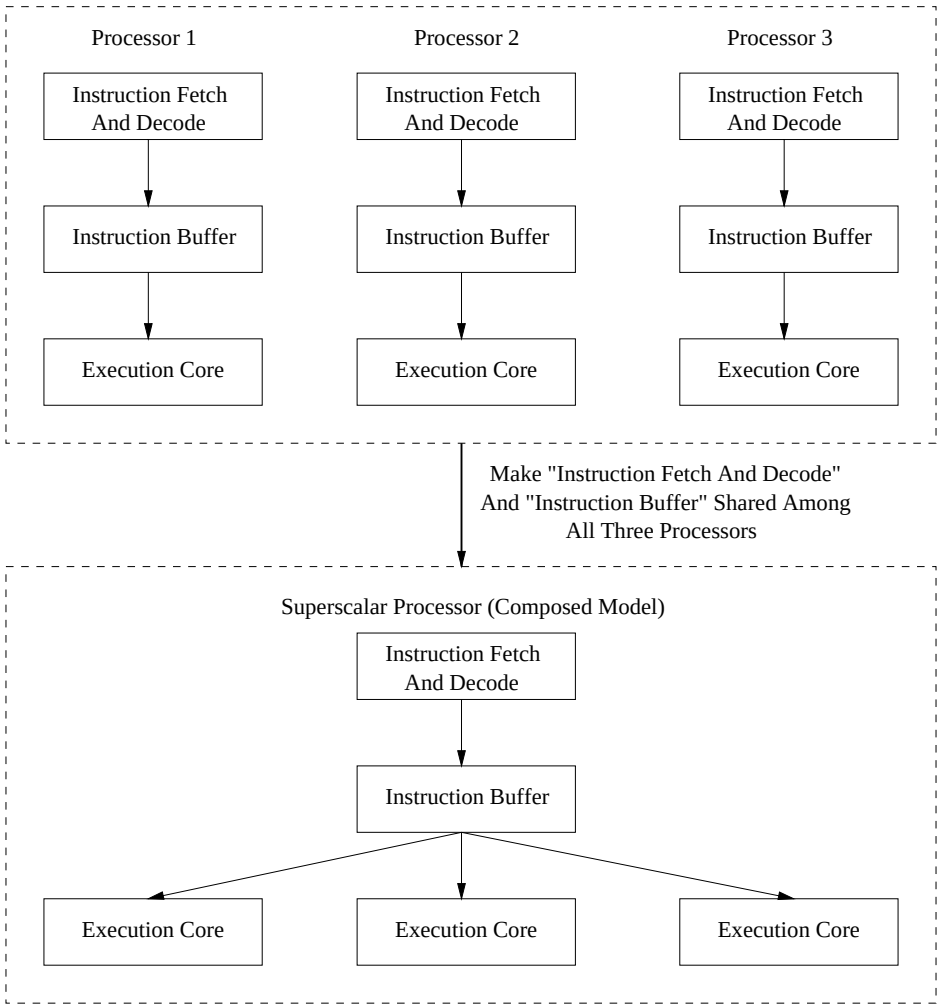


**Figure 2.1** Model Composition

decode and instruction buffer stages common, servicing all three execution cores (see Figure 2.1). Equivalence sharing provides a natural, easily understood mechanism for modeling this type of system. Later in this chapter, we will see how functional sharing techniques provide additional flexibility in the ways that the modeler may bring together atomic models and specify the interactions between them.

## 2.2 Overview of State Variables

In this section, we will introduce the fundamental elements used to represent state in Möbius, which we refer to as *state variables*. We will discuss the specific details of how state variables are constructed and how they may be used to share state between models. In the following two chapters, we will see two implementations of composer formalisms that use state variables to combine atomic models using our state-sharing framework.

In Möbius, we use state variables to hold the total state for the system to which they belong. We associate each state variable with a state variable *type*. The type of a state variable defines the domain of values that the state variable can take on. For example, an integer state variable can only hold integer values, while a floating-point state variable can be assigned floating-point values. In addition, state variables may also have a notion of structure. For example, one can construct arrays of unordered types, in which each unordered type holds several structs. The domain of state variables that can be shared across models will have the grammar shown in Figure 2.2. The recursive nature of the grammar gives great flexibility in the construction of state variables.

State variables may consist of either an ordered or unordered set of elements. With an ordered set, the ordering of the elements is significant; for example, the ordered set of letters (A, B) would not be the same as the ordered set of letters (B, A). However, with an unordered set, the elements may be permuted in any order without changing the nature of the set. This means that the unordered set {B, B, A, B, C} is valid and is distinctly different from the unordered set {A, B, C}, but is equivalent to the unordered set {A, B, B, B, C}. It should be noted that the unordered sets that we refer to here are equivalent to bags with fixed cardinality.

```
SharableStateVariable → StateVariableType
StateVariableType     → BasicType | ArrayType | StructType | UnorderedType
ArrayType             → Size = integer StateVariableType
BasicType             → CharType | IntegerType | ShortType | FloatType | DoubleType
CharType              → name CharType
IntegerType           → name IntegerType
ShortType             → name ShortType
FloatType             → name FloatType
DoubleType            → name DoubleType
StructType            → { StateVariableList }
StateVariableList     → StateVariableType | StateVariableType StateVariableList
UnorderedType         → Size = integer StateVariableType
integer               → Sequence of digits that cannot be preceded by a minus sign
name                  → Any valid state variable name
```

**Figure 2.2** State Variable Grammar

To understand how state variables are used in the construction of atomic models, consider the first atomic model formalism implemented in Möbius, the stochastic activity network formalism. In stochastic activity networks, one or more *places* will collectively contain the entire state of a given model. Each place will hold a nonnegative number of *tokens*, where the number of tokens represents the state of the place. In the *UltraSAN* [8] implementation of SANs, short integers are used to hold the number of tokens in a place. In the Möbius implementation, places are implemented as state variables with type Place. The state contained by a place can be fully represented by a short integer, and would consist of a simple **ShortType**.

With a place, we are only concerned with its current *marking*, which is a mapping from each place to a natural number [1]. However, in an implementation, the value (marking) of a place will usually be limited to that of a short integer (an integer between -32767 and 32767, inclusive). Although the negative values are not normally used in modeling, they provide a simple (partial) validation that the gate functions are as intended. If a greater range is desired,

a larger data type, such as an integer or double precision integer, could be used instead of a short integer.

An example of a state variable with structure is a MultiClassQueue type from a queueing network formalism, having the following structure:

**Size** = 1024 {

ID **ShortType**

Color **CharType**

}

The queue has 1024 slots for colored tokens, where each slot consists of a short integer to represent the token ID and a character field to represent the color. We will allow the tokens to change color over time (without this property, the Color field would be redundant).

## 2.3 Equivalence Sharing

The first method by which state is shared between models in Möbius is equivalence sharing. Two state variables that are equivalently shared will become essentially the same variable, meaning that they can be thought of as a single state variable that coexists in two or more models simultaneously. In this section, we will explain the mechanisms by which the user may initiate equivalence sharing among state variables and discuss the implementation issues surrounding equivalence sharing.

### 2.3.1 Restrictions on state sharing

Not all state variables may be shared equivalently. Clearly, the type of a state variable will place restrictions on the combinations of state variables that can be shared. In theory, one could

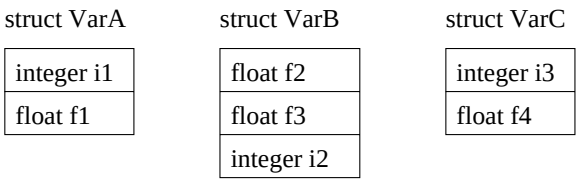


**Figure 2.3** State Variables with Structure

create a set of semantic rules to translate values between shared state variables of different types. For example, suppose we share a floating-point state variable with an integer state variable. When we look at the shared variable's state from the point of view of the floating-point variable, there will not be a problem. But when we look at the state from the integer variable's point of view, because an integer cannot fully represent an arbitrary floating-point value, we must either truncate any fractional portion or round off. However, the cost of implementing such a domain mapping mechanism must be weighed against the benefit of doing so. In practice, the likelihood that a modeler would be able to derive a substantial benefit from sharing two state variables of different types is extremely small. For this reason, we chose to avoid such a semantic mapping mechanism, thereby simplifying the implementation and speeding run-time execution of models.

Allowing state variables with structure has interesting implications for state sharing. The main issue is determining whether a state variable having structure can be shared, and if so, how it should be done. For example, consider the three structured state variables shown in Figure 2.3.

The structures of VarA and VarB are clearly different: VarB contains an additional floating-point data member and the floating point data members now come before the integer data members. However, because VarC also consists of an integer data member followed by a floating-

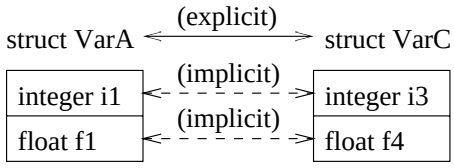


**Figure 2.4** StructA Shared with StructC Directly


**Figure 2.5** StructA Shared with StructB

point data member, VarA and VarC can be shared directly, meaning that the integer data members would be shared and the floating-point data members would be shared (see Figure 2.4). Because of their structural differences, neither VarA nor VarC can be shared directly with VarB (there is not a one-to-one, sequential mapping between the data elements). On the other hand, there is no reason why VarA's i1 and VarB's i2 cannot be shared; furthermore, it should be possible to share f1 with either f2 or f3 (see Figure 2.5). From these observations, it follows that ordered, structured state variables can be shared directly if the ordering, number, and types of their data members are identical. If any of these conditions cannot be met, their data members may be shared individually, under the same restrictions. If a structured state variable has other structured states within it, the procedure for determining compatibility between the data members can be applied recursively.

Now consider the case of the unordered set state variable type. With an unordered set, the number of elements remains constant and all of the elements must be of the same type. These restrictions are necessary in order to allow an efficient physical implementation in which unordered set state variables can be shared. If the number of elements was not constant, then a less efficient data structure, such as a linked list, would be required to realize the variable-size set. Requiring all of the elements to be of the same type simplifies the implementation. These restrictions lead to a simple rule for determining whether two unordered set state variables may be shared: they may be shared if and only if the element types and cardinalities of the sets are identical.

### 2.3.2 Implementation

Suppose we have two state variables, $A$ and $B$. These state variables exist in separate atomic models and are each affected by an action, as shown in Figure 2.6. When state variables $A$ and $B$ become shared, they can then be treated as a single new state variable $C$ that is affected by all actions that affect $A$ or $B$, as shown in Figure 2.7. For example, if the firing of an action changes the state of variable $A$ in a certain way, the firing of that action will affect the state of variable $C$ in the same way. Additionally, enabling conditions dependent on the state of either variable $A$ or $B$ will now depend on variable $C$ in the same fashion.

Ideally, when two state variables are shared, they should reside in the same location in memory. This would allow the most natural and efficient representation of the model in computer memory: all models using the shared state variable can simply write to or read from the same location. Having a common memory location reduces the memory space requirements considerably, providing an incremental benefit with each additional state variable that is shared.


**Figure 2.6** Before State Sharing

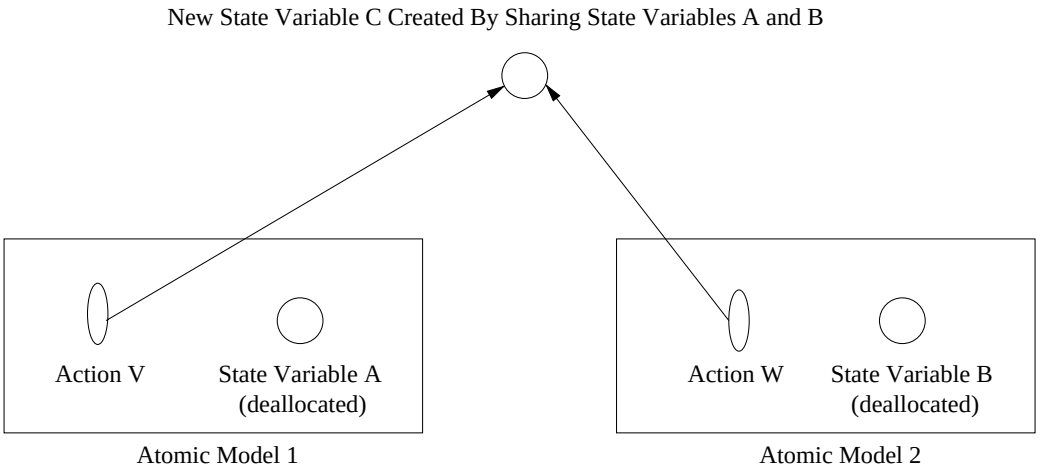


**Figure 2.7** Composed Model after State Sharing

Unfortunately, because of other design objectives in the Möbius modeling framework, we were not able to realize this ideal in the implementation. The main complication was the desire to share state variables which have different names between different formalisms. Suppose we have two atomic model formalisms: a stochastic activity network (SAN) formalism and a wide-area network (WAN) formalism. Additionally, imagine that SANs have short integer state variables named "Places" and that the WANs have short integer state variables named

"ActiveUsers." Because the two state variables are of the same type, there is no reason why it should not be possible to share them, even if their names are different. However, Places and ActiveUsers may use dissimilar methods for viewing or changing their current state.

From an object-oriented programming standpoint, the natural design for state variables is to have a base state variable class from which all other formalism-specific state variable types (like Places and ActiveUsers) will be derived. Having a base state variable type allows the formalism designer to define a state variable's formalism-specific methods and behavior in a generic fashion, providing a mechanism for model composition between different formalisms. However, instances of the different derived state variable types cannot reside at the same physical memory location. Therefore, we chose to have each derived state variable type include a reference to a separate memory location that will hold the state. When a state variable is shared with another state variable, the reference in one of the state variables will be changed to point to the same location pointed to by the other, and the unneeded memory will be freed, as shown in Figure 2.8. In this manner, each formalism-specific state variable can have its own name and methods, but the memory location holding the state can be shared.

When a model is composed, it can be connected to other models through state sharing. In our implementation, state variables can be joined even if they have different names in their respective models. Additionally, a shared state variable can be given a new name in the composed node. To achieve this functionality, we chose to physically create a new state variable in the composer node having the new user-defined name, and to share the submodel's desired state variables with it. Only the composer node's newly created state variables are visible outside of the composer node.

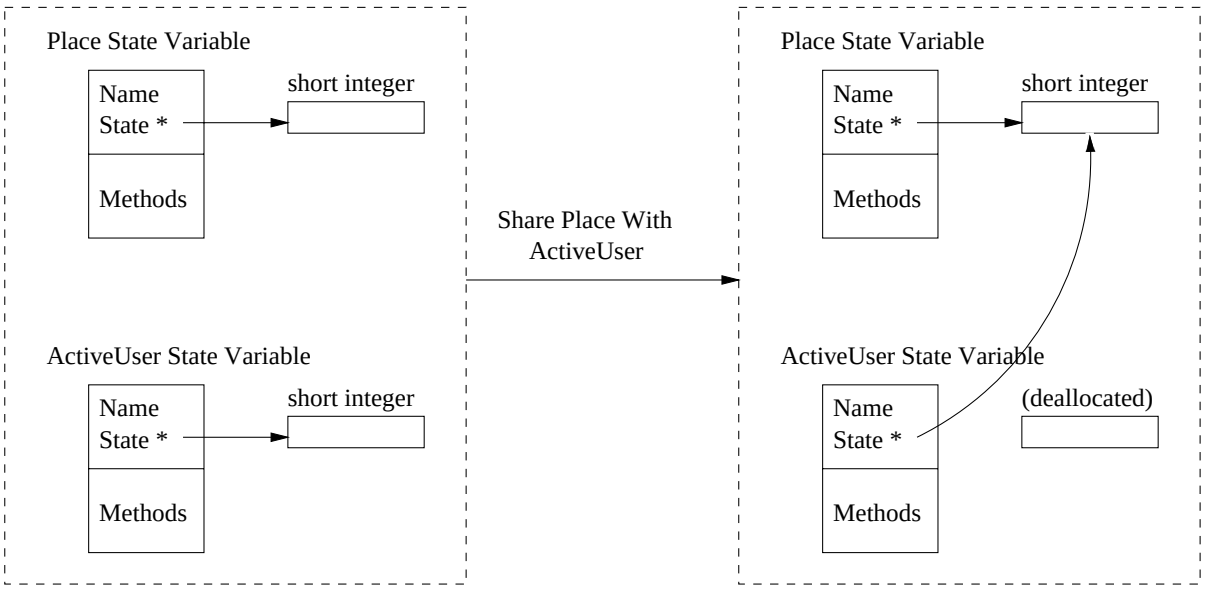


**Figure 2.8** Place Shared with ActiveUser

The composer node will maintain a list of the newly created state variables; additionally, it will maintain a list of the submodels' state variables that were shared in creating the new state variables. This is necessary because of a speed optimization that was done to increase the speed of simulation: any model actions that previously affected a submodel's shared state variable will now affect only the newly created state variable in the composer node. This optimization reduces the amount of work that must be done when an action fires. However, this increases the amount of bookkeeping and initialization work that must be done by the composed node. For example, if the user wants to define a reward variable on a specific state variable in one of a composer node's submodels, the composer node must intercept the reference to the submodel's state variable and internally change it to refer to the newly created state variable.

## 2.4 Functional Sharing

In addition to the equivalence method of sharing state variables, Möbius can provide a mechanism for allowing the modeler to associate the value of state variables with functions. The functions may be an arbitrary function that can be expressed in C++, including functions dependent on the value of other state variables. The only restriction is that the function's return value must be of the same type as the state variable that it is shared with.

State variables that are shared with functions cannot be treated like equivalently shared state variables; specifically, state variables shared with functions cannot be assigned values. This is because a functionally shared state variable will always assume the value returned by the function, meaning that arbitrary values can no longer be assigned to that state variable. For example, if we have a state variable that is shared with a function that returns the length of a particular queue in our model, then we can only read the state of that variable; we cannot set the state of the variable directly. In other words, the functionally shared state variable is *read-only*.

### 2.4.1 Implementation

Providing a mechanism for functional state sharing in the implementation required some changes from the equivalence sharing mechanism. With equivalence sharing, the state variables can be simply instantiated, shared, and accessed through the shared memory location. Functional sharing does not make use of a shared memory location, meaning that functionally shared state variables, when they are accessed, must be able to call the function with which they are shared. The main design issues involved deciding where to declare the shared functions and how the functionally shared state variables would access them.

It would be ideal if the function to be shared only needed to be declared once, and state variables needing to call the function could reference it. However, although the model would execute faster if each state variable had a local declaration of the function, the local declarations would not be desirable from an object-oriented design standpoint. The atomic models are meant to be self-contained and should have knowledge neither of other atomic models nor of the composed models that include them. In order for them to declare the functions locally, they would need to query the composed models that include them in order to determine what functions to declare. Additionally, they would need to obtain references for any data members that need to be accessed to compute the value of the function. Not only would this organization significantly increase the complexity of the atomic interface, but it would also create some ambiguity if an atomic model were to be included in multiple, separate composed models.

In order to simplify the implementation and to create a clean, object-oriented design, the following design was adopted:

- The user will specify the functions to be shared and the state variables that will be shared with them in the composer interface.
- The functions will be declared in the C++ code outputted by the composer interface.
- At model initialization time, the composer nodes will instantiate the submodels they include and initiate the functional sharing, passing a function pointer referencing the new function to the functionally shared state variables.

The code shown in Figure 2.9 demonstrates how a read-only state variable for use with functional sharing can be defined. The class contains a pointer to a function that will be initialized later

```
class ReadOnlyStateVariable:public BaseStateVariable
{
 public:
  ReadOnlyStateVariable();
  ~ReadOnlyStateVariable();

  short (* getState) ();
  void setStateToSharedFunction(short (* function) ());

 private:
  short InvalidDefinition();
}

ReadOnlyStateVariable::ReadOnlyStateVariable() {
  getState = &ReadOnlyStateVariable::InvalidDefinition;
}

void ReadOnlyStateVariable::setStateToSharedFunction(short (* function) ()) {
  getState = function;
}

short ReadOnlyStateVariable::InvalidDefinition() {
  cerr << ``Read-only state variable not defined!'' << endl;
  return -1;
}
```

**Figure 2.9** A Read-Only State Variable Class

by the composed model constructor. In case the function is never initialized and a call is made to get the state, an error message will be displayed.

In our implementation, allowing the user the flexibility of specifying custom functions to compute the values of state variables leads to a performance penalty over simple equivalence sharing. This penalty stems from the fact that the functionally shared state variables need to dereference a function pointer to compute their current value. While the incurred overhead amounts to no more than a single level of indirection, it is still desirable to avoid this overhead when doing normal equivalence sharing. Unfortunately, this means that the state variable will need to be declared differently so that it will directly reference its state, instead of dereferencing

a function pointer to obtain its value. Because atomic models are not allowed to query composed models to see whether they have functionally shared state variables (for the reasons stated above), we require the user to declare state variables to be read-only within the atomic model interfaces. If a state variable is declared read-only, the atomic model interface will prevent the user from defining the model in such a way that the state of the read-only state variable can be modified locally. This may be viewed as an additional burden on the modeler, but it forces the modeler to plan ahead and prevents him or her from creating a model that will have illegal semantics after it is composed and its state variables are functionally shared.

### 2.4.2 Example

In this example, we will see how atomic models from two different formalisms can be joined using functional sharing. We will take a queueing network model and connect it to a stochastic activity network model. In doing so, we will demonstrate how functional sharing can be used to specify more intricate model-to-model interactions than are available with equivalence sharing techniques. We expect the importance of functional sharing to grow as more and more atomic and composer formalisms are integrated into the Möbius modeling framework.

Consider the queueing network shown in Figure 2.10. The example queueing network atomic model contains three queues: Resources Available, Resources to be Recycled, and Consumed Resources. Execution of the model begins with a nonnegative number of resources in the Available Resources queue. Available Resources, upon usage, may go into a waiting queue to be recycled, or may simply become waste and move to the Consumed Resources queue. Resources to be recycled may either be successfully recycled and returned to the Available Resources queue, or be moved to the Consumed Resource queue if the recycle process fails.

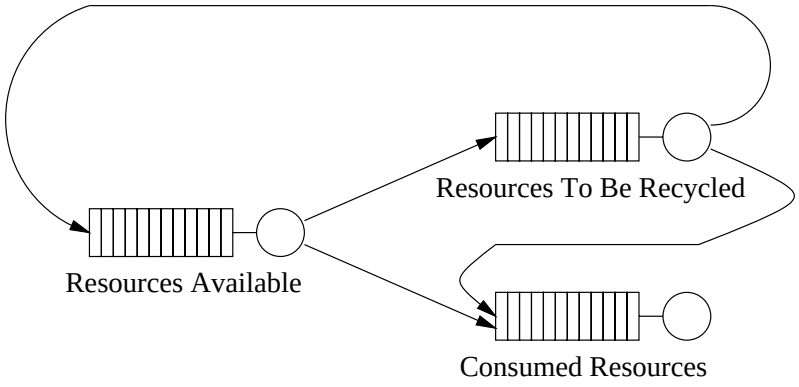


**Figure 2.10** Queueing Network Atomic Model

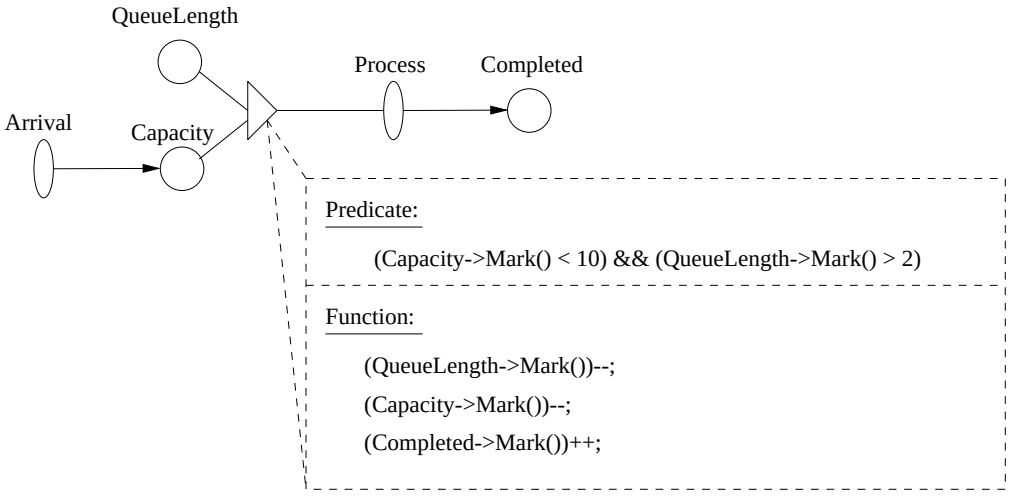


**Figure 2.11** Stochastic Activity Network Atomic Model

The base Queue class, from which all three of these example queues are derived, has a function *queueLength()* that returns a short integer denoting the number of elements currently in the queue.

Next, consider the stochastic activity network atomic model shown in Figure 2.11. In this model, there are three places: QueueLength, Capacity, and Completed. Capacity indicates the

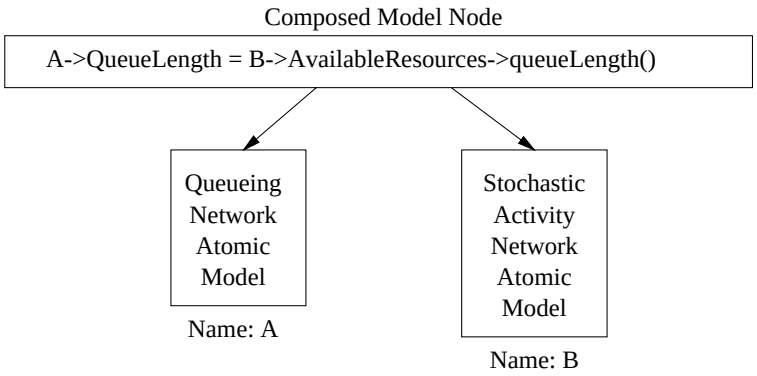


**Figure 2.12** Composed Model Demonstrating Functional Sharing

maximum number of items that the input buffer can hold. The number of tokens in Completed shows how many items the system has processed. QueueLength will be used to indicate how many items are in the Available Resources queue of the queueing network atomic model. The input gate prevents the system from doing any processing if the system is over capacity or if there is an insufficient number of elements in the Available Resources queue.

We then build a composed model from the queueing network and stochastic activity network atomic models and show how functional sharing can be used to connect the models. The marking of the QueueLength place in the stochastic activity network atomic model is set to the value of the current queue length of the Available Resources queue in the queueing network atomic model. The composed model is shown in Figure 2.12.

As mentioned previously, the user must declare a state variable to be read-only before it can be functionally shared. Since we implement places from stochastic activity networks as place state variables in Möbius, we will need to derive a read-only place state variable class from the base read-only state variable class shown in Figure 2.9. Using the read-only place state

```
class SampleSAN:public SAN
{
  public:
  SampleSAN();
  ~SampleSAN();

  ReadOnlyPlace * QueueLength = new ReadOnlyPlace();

  // ... other data members and methods ...
}
```

**Figure 2.13** SAN Model Having a Read-Only Place

variable, we can create places to be used for functional sharing in stochastic activity network atomic models, as seen in Figure 2.13.

Now that we have an atomic model that is ready to be functionally shared, we will compose it with another atomic model and initiate functional sharing between the two. Assume that we also have a class SampleQueueingNetwork which defines the queueing network atomic model that we have described. The composed model would then initiate the functional sharing between place QueueLength in the SAN and a function returning the length of the Available Resources queue in the queueing network model, as shown in Figure 2.14. The composed model first creates the two atomic models. The composed model also defines the function that retrieves the length of the Available Resources queue from the queueing network model. After the function has been defined, the composer constructor will initialize the getState function pointer for the read-only place in the SAN.

## 2.5 Summary

In this chapter, we have introduced the mechanism in Möbius by which the user can bring together atomic models to form composed models. Specifically, we use state variables that

```
class SampleReplicateJoin:public ReplicateJoin
{
  public:
  SampleReplicateJoin() {
  // Instantiate the atomic models to be composed.
  SampleSAN * MySAN = new SampleSAN();
  SampleQueueingNetwork * MyQueueingNetwork = new SampleQueueingNetwork();

  // Define the new function to be shared with place QueueLength.
  short getAvailableResourcesLength() {
  return MyQueueingNetwork->AvailableResources->getQueueLength();
  }

  // Initiate the functional sharing
  MySAN->QueueLength->setStateToSharedFunction(&getAvailableResourcesLength);

  // ... Other state sharing directives ...
  }
}
```

**Figure 2.14** A Replicate/Join Composed Model with Functional Sharing

can be constructed and shared between models in a flexible fashion. We also have seen how the state-sharing framework in Möbius allows for two methods of model connection via state sharing: equivalence sharing and functional sharing. With equivalence sharing, two or more state variables are shared by using a common memory location. With functional sharing, a function of one state variable defines the state of another state variable in a different model. Finally, we have discussed some of the implementation issues that we encountered in developing the state-sharing framework, and have explained how they were addressed.

The state-sharing framework is general enough to allow the modeler great flexibility in specifying model-to-model connections and interactions; it also facilitates the creation of new formalisms better suited to his or her specific modeling needs. In the next two chapters, we will see two implementations of composer formalisms that make use of the state-sharing framework presented in this chapter.

# CHAPTER 3

# REPLICATE/JOIN COMPOSER FORMALISM

## 3.1 Introduction

In this chapter, we will present the first composer formalism to be implemented on top of the Möbius state-sharing framework, the Replicate/Join formalism [9]. After describing the theory behind the state reduction techniques used in this composer formalism, we show a sample model and demonstrate how state spaces for Replicate/Join composed models grow at a much slower rate than those of noncomposed models. We also discuss many of the design decisions made in the Möbius implementation of the Replicate/Join formalism. Finally, we explain how model validation is performed for Replicate/Join composed models in our implementation.

## 3.2 Theory

In order to perform mathematical analysis of a model, it usually is necessary to generate a state space for the model. Unfortunately, for nontrivial models, the full state space will often be prohibitively large. Not only will the computer memory requirements be excessive, but the time needed to either generate the state space or solve the underlying Markov chain will make the exercise a practical impossibility. Modern design processes normally require several iterations through a series of design steps, with the design being improved across successive iterations. Analytic solution and simulation will usually be one of the final steps in the design flow, but the next design iteration most likely will depend upon the data they provide. If analytic solution

alone takes several days or weeks to complete, it can become a critical design bottleneck; for this reason, mathematical solutions are often omitted, and simulation will be used exclusively.

Unfortunately, the complete omission of mathematical analysis from the design cycle is not without cost. Mathematical analysis is preferable to simulation for certain aspects of modeling, such as rare events. For example, the chances of an asteroid colliding with a satellite may be quite small, but it may be important to model this event in order to determine the system's ability to handle a critical failure. This type of modeling can be difficult to do accurately with simulation, even if the simulator is modified to handle rare events (also known as *importance sampling* [10]). For these situations, traditional mathematical solution techniques will still provide the most trustworthy and accurate results.

However, as mentioned previously, traditional mathematical solution techniques rely upon the generation of a state space, followed by solution of the underlying continuous-time Markov chain (CTMC). In order to make the solution of the CTMC feasible in terms of required computer memory and processing time, we need to reduce the size of the state space. In general, there are two ways to achieve smaller state spaces:

(1) Generate the full state space, and then apply state lumping techniques to eliminate redundant states.

(2) Observe the structure of the CTMC first, and then build a smaller state space directly.

With the first approach, the original problems of computer memory and processing overhead have not been solved; in fact, one could argue that they have only been exacerbated by the addition of a compression stage after the initial state-space generation. For this reason, the second approach is preferable, as long as the observation stage is reasonably efficient. The Replicate/Join composition formalism takes this approach by utilizing the strong lumping the-

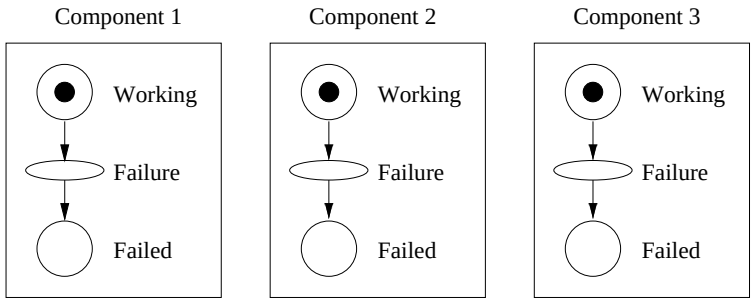


**Figure 3.1** Three Components with Exponential Rates of Failure

orem [11]. The strong lumping theorem is the result of work by Kemeny and Snell in the 1960s. It exploits symmetries in the state space of a Markov process to generate a new, smaller state space that is still a Markov process. Formally, the theorem is as follows:

**Theorem 1** *Suppose we have a continuous-time, discrete-state Markov chain (CTMC) and an equivalence relation $f$ that partitions the state space. Then $f(X_t)$ is a Markov process, if for every pair of equivalence classes $G$, $H$ $(G \neq H)$ of $f$, the rate for every $X \in G$ to $H$ is identical.*

For example, suppose we have a CTMC which models the reliability of three components. The stochastic Petri net representation of the three components is shown in Figure 3.1. Suppose that all three components are working initially, and that each component fails with rate $\lambda$. The graphical representation of the state space is shown in Figure 3.2.

By applying the strong lumping theorem, we can reduce the state space to four equivalent states with new rates, as shown in Figure 3.3. Instead of maintaining information regarding exactly which components are working or not working at a particular time, we just keep track of how many components are working or not working. For example, consider the two system states shown in Figures 3.4 and 3.5. In both states, one component has failed; however, we now treat the two states as *equivalent*. In other words, we *lump* the states. While we no longer have

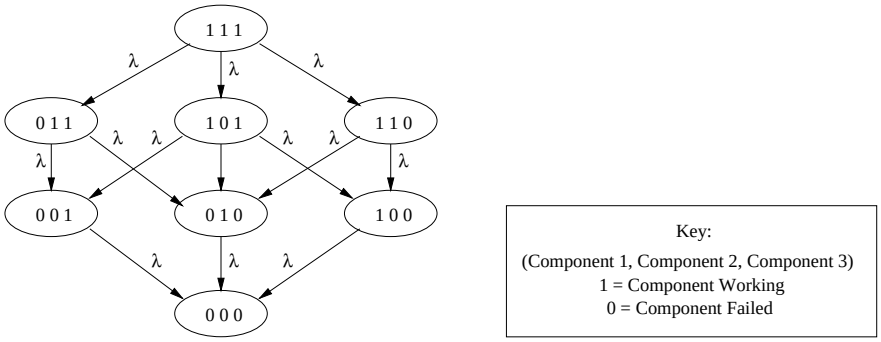


**Figure 3.2** Three Component-Reliability State Space

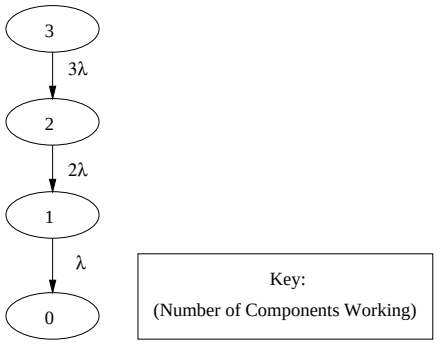


**Figure 3.3** Three-Component Reliability Lumped State Space

information regarding the independent behavior of components, this is not a problem because all of the components will behave identically and we are more concerned with the reliability of the system as a whole.

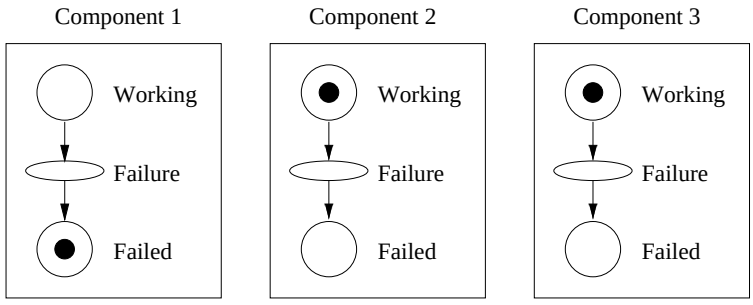


**Figure 3.4** Three Components with Failed Left Component

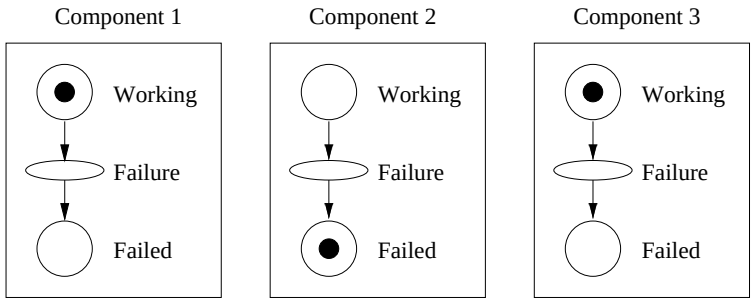


**Figure 3.5** Three Components with Failed Middle Component

## 3.3 Replicates and Joins

The Replicate/Join composition formalism provides a hierarchical, tree-like method of combining atomic models to form larger, composed models. As the name may imply, there are two main methods that may be used to combine models: Replicates and Joins. Replicates are used to replicate a model any number of times, often having one or more state variables shared among all of the replicas as a means of connection. Joins are used to bring together two or more dissimilar models, connecting them by sharing certain state variables between them. We will refer to a model being replicated or joined as a *submodel* of the Replicate or Join including it. Similarly, we will refer to the Replicate or Join including the submodel as the *parent* of the submodel. Each submodel, whether it is one of many replicas or a unique submodel in a Join, will have its own state.

To illustrate how the Replicate/Join formalism may be used to build a model having symmetry, we use an example model. We will construct a simplified model that can be used to analyze the outgoing passenger flow in a typical airport, using the stochastic Petri net formalism for the atomic models. Assume that the airport is laid out as shown in Figure 3.6. We begin with a nonnegative number of outgoing passengers at the main airport entrance. From

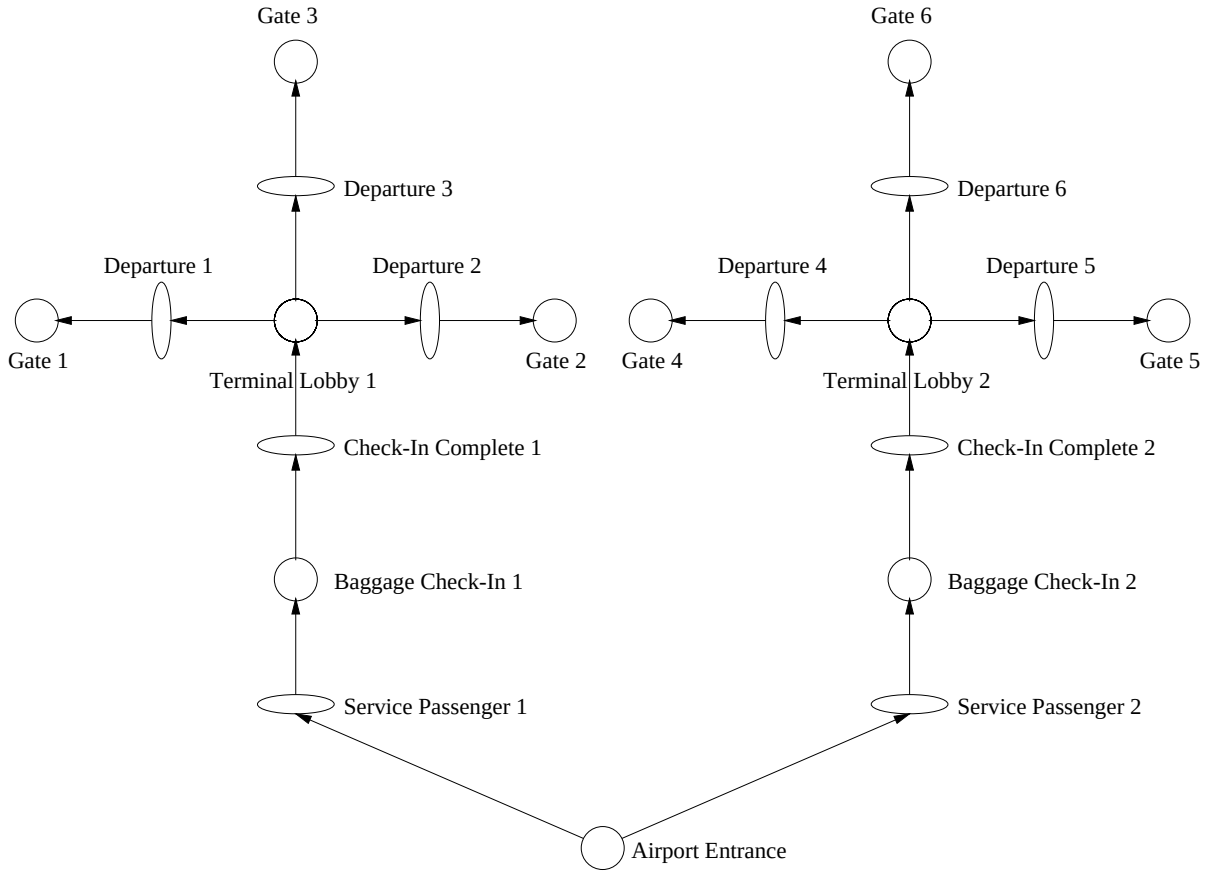


**Figure 3.6** Simple Airport Model

there, they proceed to the baggage check-ins for the airlines that they will be flying on. After checking in their baggage, they proceed to their airlines' lobbies, where they await the arrival of the planes that they will be flying on. After each plane arrives and has been refueled and serviced, the appropriate passengers will depart through a gate and board the plane.

Clearly, our model has symmetry. Each airline's terminal has several identical gates. At a higher level, each airline's combination of baggage check-in, lobby, and gate cluster is identical to that of the other. As we will see, the Replicate/Join formalism is well-suited to modeling systems that have built-in symmetry, like our airport.

To convert the flat model shown in Figure 3.6 to a hierarchical Replicate/Join tree, we will need to hierarchically compose the atomic models shown in Figure 3.7. We will use Replicate

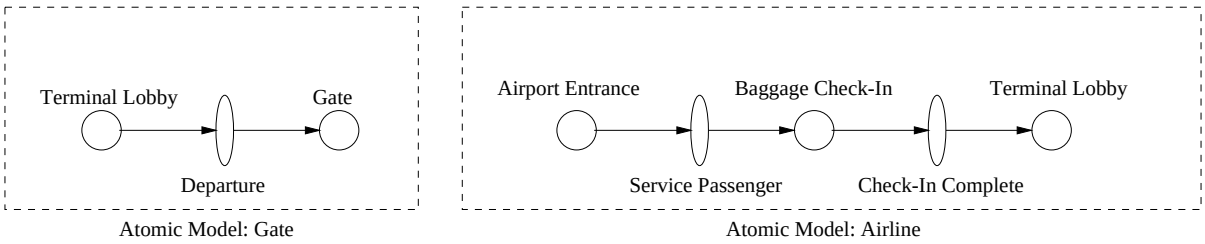


**Figure 3.7** SPN Atomic Models Used to Construct Airport Model

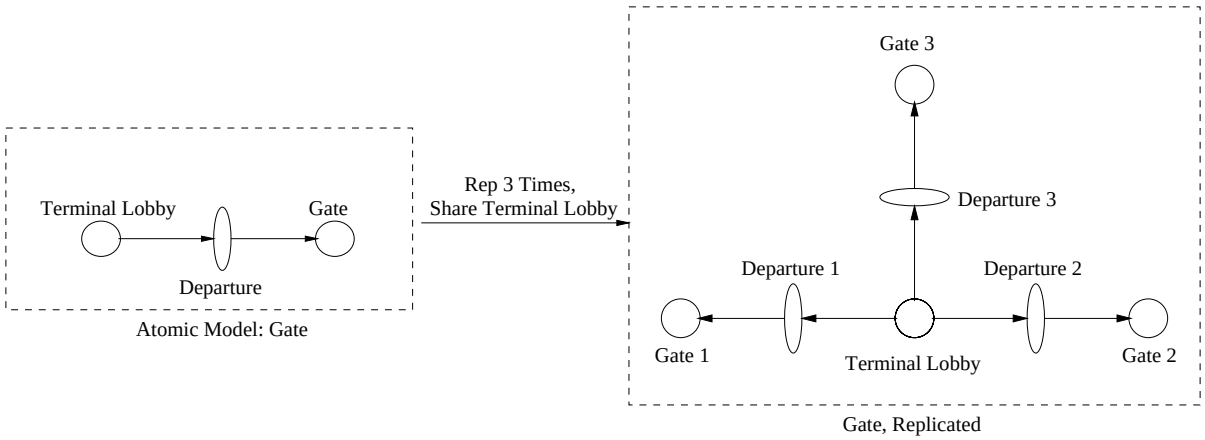


**Figure 3.8** Replicated Gates with Shared Lobby

nodes to represent the parts of the system that have symmetry. The terminal gates are represented by a Replicate node replicating the gates the necessary number of times, and connecting the replicas by designating the lobby to be common (as seen in Figure 3.8). After constructing the gate cluster, we connect it to the rest of the airline's components by means of a Join, as shown in Figure 3.9. Now that we have the airline fully constructed, we can model the airport as a whole by using a Replicate node to create several instances of the airline, connecting them at the main airport entrance. Construction of the final composed model is demonstrated in Figure 3.10, and its Replicate/Join tree representation is shown in Figure 3.11.

The Replicate/Join formalism's hierarchical method of combining models is easy for modelers to learn and understand. For several years, design engineers and other modelers have made frequent use of it in designing complex, dependable systems. For example, engineers at Mo-

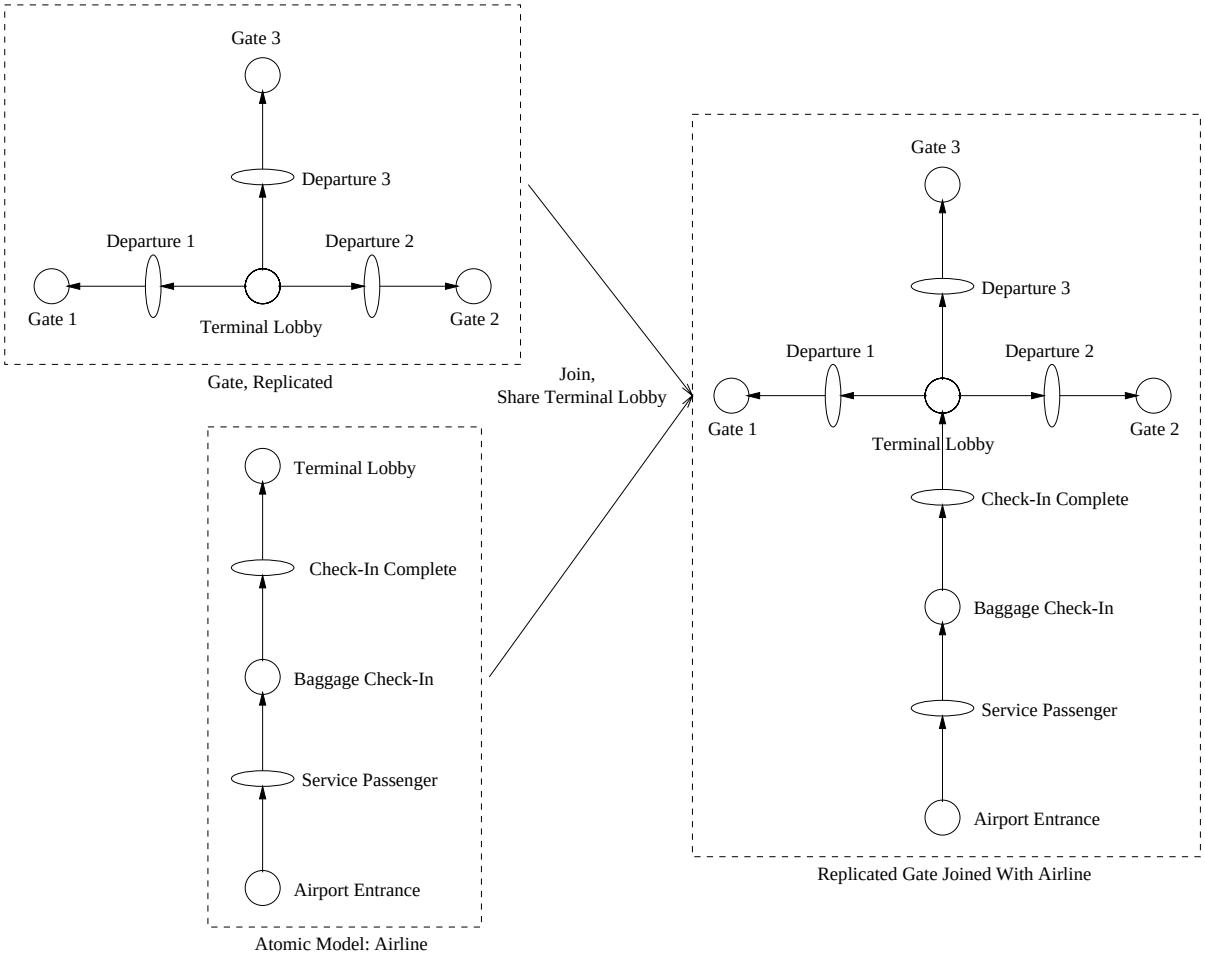


**Figure 3.9** Replicated Gates Joined to Airline

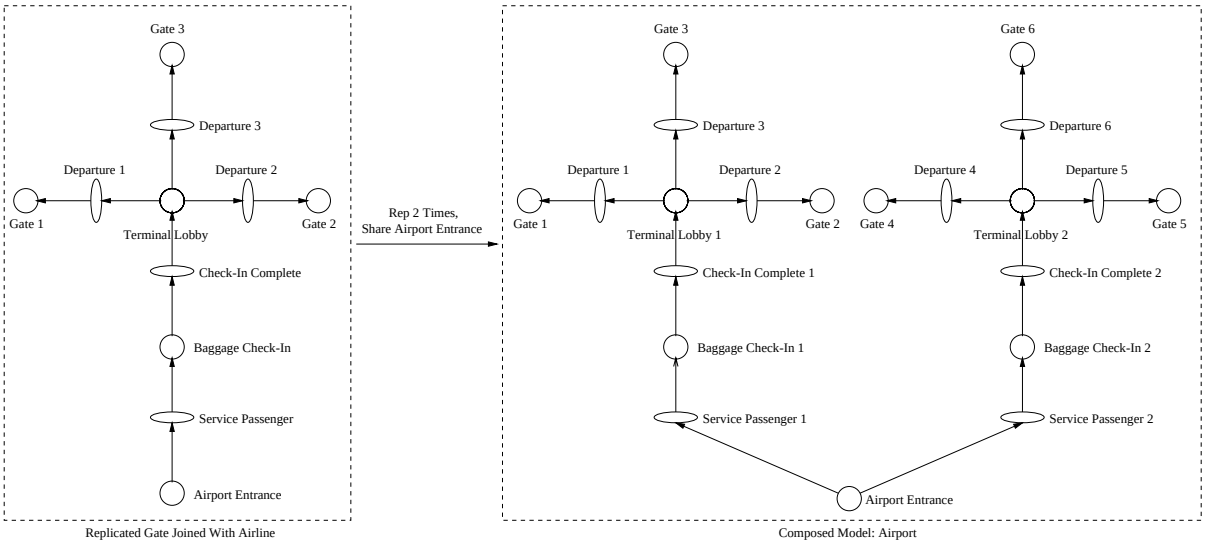


**Figure 3.10** Construction of Final Composed Airport Model

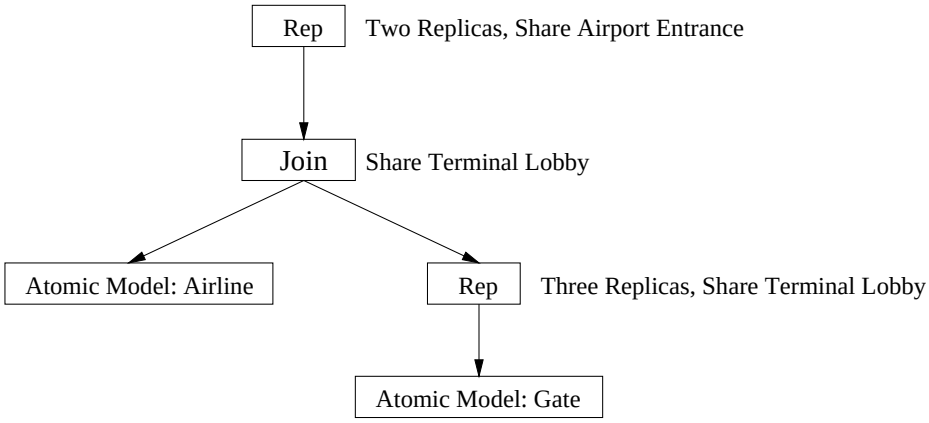


**Figure 3.11** Replicate/Join Tree Representation for Airport Composed Model

**Table 3.1** Comparison of State Spaces for Flat Model and Replicate/Join Tree

| Initial Markings | | Number of States | |
|---|---|---|---|
| Airport Entrance | All Other Places | Flat Model | Replicate/Join Model |
| 1 | 0 | 11 | 4 |
| 5 | 0 | 3003 | 361 |
| 10 | 0 | 184756 | 12269 |
| 15 | 0 | - | 155874 |

torola use it extensively in the design and analysis of their satellite constellations. Another key advantage of the Replicate/Join formalism is that it will generate reduced state spaces. The use of Joins in a composed model will not yield a smaller state space, but Replicate nodes apply the strong lumping theorem to reduce the size of the composed model's state space. Specifically, a Replicate node will only keep track of the number of its submodels that have a particular state, rather than keeping track of the state of each submodel. Table 3.1 shows how state lumping reduces the number of states in our airport model.

## 3.4 Implementation

Within Möbius, the Replicate/Join formalism's functional implementation attempts to achieve efficiency in terms of both speed and memory consumption. These goals are, to some extent, mutually exclusive, so some tradeoffs needed to be made in the design. In this section, we will discuss the main design decisions that we encountered and how they were addressed. The key design decisions that we will discuss are

- how Replicate nodes and state lumping fit in with the Möbius modeling framework;
- how to implement state lumping techniques to achieve smaller state spaces, and when the techniques should be applied; and
- how to represent the state of Replicate nodes.

### 3.4.1 Replicate nodes within the Möbius environment

The first main tradeoff in the design involved deciding how the different components of the Möbius modeling tool will interact with Replicate nodes, and how the state lumping techniques should be applied. Replicate nodes apply the strong lumping theorem, in that they track the number of submodels that have a given state, instead of reporting the state of individual submodels. What this means in our implementation is that a Replicate node considers the state of each submodel and sorts the states into bins of unique states, where each bin has an associated state cardinality indicating how many submodels have that unique state. The bins collectively hold the state for the entire composed model. We then need to decide how the state information contained in the bins can be used by other parts of the Möbius tool. As we have designed it, our state lumping procedure converts the internal representation of a

Replicate node to a consistent external representation of state to be used by solvers, state space generators, or other models. The details of this procedure will be explained below.

For the purpose of reducing the state space, there needs to be a way for a state space generator to determine whether two states are equivalent: either the state space generator needs to have a method of comparing two lumped states, or the composed model needs to handle all state comparisons. This is because the state space generator will, in traversing the complete state space, make repeated calls to the model to check whether a state has already been encountered, by comparing the current state to previously encountered states. Ideally, the state space generator will not need to have detailed information of how models in particular formalisms are constructed, so we decided to delegate the responsibility for determining state equivalence to the Replicate nodes themselves. This means that the state space generator can simply ask a Replicate node whether two states that it is currently considering are equivalent; if they are, it will not be necessary to allocate a new state in the state space. For reasons that will be explained in more detail later, we decided to have Replicate nodes report their states in a consistent external format, so that states can be compared with each other via a simple memory comparison.

### 3.4.2 State lumping within Replicate nodes

The second main design consideration that we will discuss revolves around the mechanism by which a Replicate node determines whether two lumped states are equivalent. The state of a Replicate node is equal to the union of the states of its submodel replicas. The *lumped state* for a Replicate node is slightly different in that it consists of a set of (replica state, state cardinality) pairs. So if we have a Replicate node with five replicas, where three of the replicas

have state $S1$ and the other two have state $S2$, the lumped state of the Replicate node would be the set $\{(S1, 3), (S2, 2)\}$. In theory, the lumped states for two Replicate nodes are equivalent if both of these conditions are satisfied:

(1) There is a one-to-one correspondence between the unique submodel states in the two Replicate nodes.

(2) The cardinality of each unique submodel state in the first Replicate node matches the cardinality of the corresponding unique submodel state in the second Replicate node.

In order to lump the states, we use a *translation* algorithm to construct an external representation of state. The basic translation algorithm is as follows:

(1) Begin with an empty set $S$, which will hold the pairs.

(2) Generate the first pair: (state of first replica, 1). Add it to the set $S$.

(3) For the next replica, see if it matches the first element in any of the pairs in $S$. If it does, increment the state cardinality (the second element) in the matching pair. Otherwise, generate a new pair: (state of current replica, 1). Add the new pair to $S$.

(4) Repeat Step 3 for each remaining replica.

(5) Perform an insertion sort on the pairs so that the pairs are ordered lexicographically by the first element in each pair, in order to provide a consistent external representation of state.

In order to alleviate the problem of wasted work during the translation algorithm, we added a simple cache to remember the most recent unsorted and sorted states. In other words, the

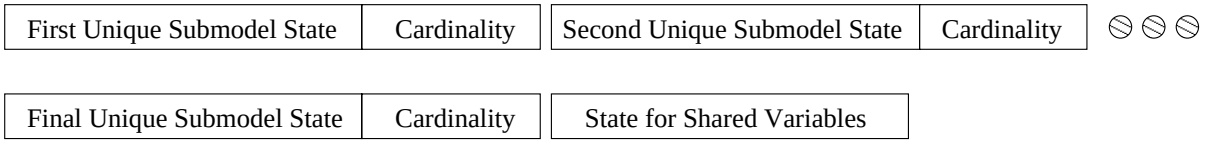


**Figure 3.12** Initial State Representation for Replicate Nodes

Replicate node will remember the state of its submodels and shared state variables before the translation algorithm, and will also remember the state to which it returns after the sorting procedure is carried out. The next time a Replicate node is asked for its state, if the Replicate node looks at the state of its submodels and shared state variables, and all of their states are the same as before, it can avoid carrying out the translation algorithm again, and simply return the same answer as before.

### 3.4.3 External representation of state

The final tradeoff in the design that we will discuss has to do with the external representation of state for Replicate nodes. We chose to use a more compact representation, rather than a larger, slightly faster representation. Recall the fact that Replicate nodes only keep track of the number of submodels that have a given state, rather than tracking the state of each submodel separately. This initially led us to design the format shown in Figure 3.12 for the internal representation of state at a Replicate node.

However, in order to reduce the amount of memory required to hold a given model's state (and the memory size of the resultant state space), we chose to place all the model cardinalities together and to compact their representation. The compaction algorithm involves bit packing and is done differently for different numbers of submodel replicas. The smallest possible representation of state cardinalities will be for one to three replicas; with this arrangement, only two bits are needed to represent each state cardinality, because in the worst case all three

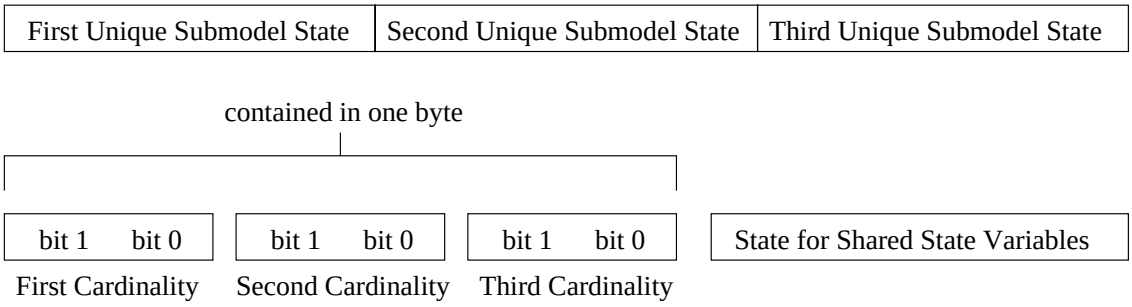


**Figure 3.13** State Representation for One to Three Replicas

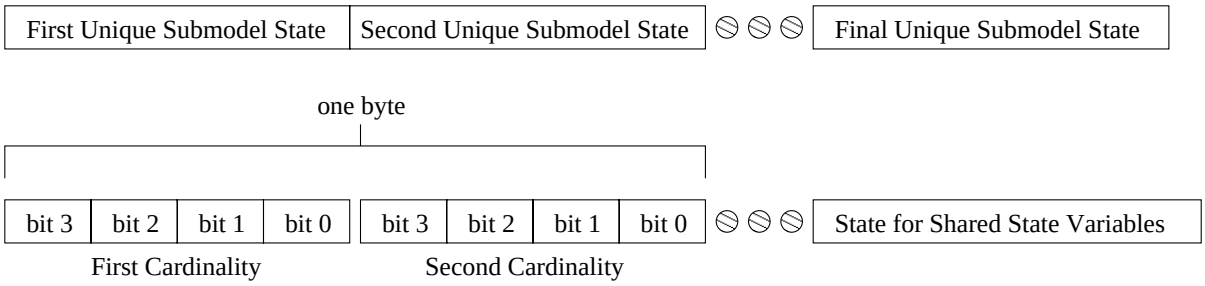


**Figure 3.14** State Representation for Four to Fifteen Replicas

submodels will have the same state. This means that for one to three replicas, the entire state cardinality portion of state can be contained in a single byte (see Figure 3.13). When we have four to fifteen replicas, up to fifteen replicas may have the same state, so four bits are needed to represent each state cardinality. This means that between two and eight bytes will be needed to represent all of the state cardinalities, as shown in Figure 3.14. Similarly, sixteen to 255 replicas' state cardinalities each require a full byte, requiring a total of sixteen to 255 bytes of memory for all of the state cardinalities. 256 to ($2^{16}$ - 1) replicas will require two bytes for each state cardinality. Finally, $2^{16}$ to ($2^{32}$ - 1) replicas will require four bytes to represent each state cardinality. More than ($2^{32}$ - 1) replicas are not supported in the Möbius implementation of the Replicate/Join formalism because of the lack of a sufficiently large native numerical data type in C++.

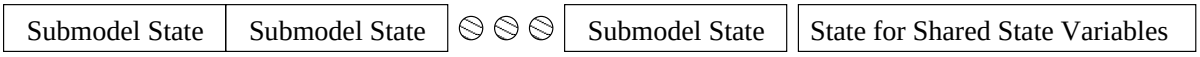


**Figure 3.15** State Representation for Joins

The external representation of state for Joins uses a more straightforward format, illustrated in Figure 3.15. The state of each joined submodel is enumerated sequentially, followed by the state(s) of the shared state variable(s). Because Joins do not apply the strong lumping theorem, they do not contribute to any reductions in the size of the composed model's state space.

### 3.4.4 C++ classes

Our C++ implementation of the Replicate/Join composer formalism consists of three main classes: BaseComposerClass, Join, and Replicate. BaseComposerClass is the main class from which Join and Replicate are derived. It was convenient to define a base class because the initialization and internal representation of Replicates and Joins have much in common.

#### 3.4.4.1 Base Composer Class

BaseComposerClass is derived from BaseModelClass, the class from which all atomic models are also derived. The BaseComposerClass constructor handles the parts of composer node intialization that are common to Replicates and Joins; for example, it sets up the actions and groups for the composer node. Methods common to both Replicates and Joins will be handled by this class. However, Replicates and Joins will overload the methods that need to be handled differently, such as fetching the current state.

**Model Array** Each Replicate or Join will have an array of pointers to either atomic or composed models. For a Replicate node, this will be an array of models with identical structures, but possibly different current states. For a Join, this will be an array of the models that have

been structurally joined. Each pointer in the array will point to an individual instance of the model. These submodels are instantiated when the composer node is instantiated.

**Model Actions** Each Replicate or Join contains an array of pointers to the actions associated with it. The set of actions associated with a composed node is the combination of all the actions in each of its submodels. Even if a submodel is replicated, each replica will have its own individual set of actions that will, in turn, be associated with its parent Replicate node. For example, if a given atomic model has two actions and it is replicated five times, then its parent Replicate node will have ten actions associated with it.

**Model Groups** Each action will belong to a group that defines its execution policy. An action group is a collection of one or more actions, from which only one representative action can be fired at any state space generation or simulation step. The group representative is chosen from among the set of enabled actions, based upon their *rank* and *weight*. The rank of an action determines which action will be selected for firing from among the set of currently enabled actions at a given time. Only the highest-ranking actions may be fired in the current state. If more than one action has the same rank as the highest-ranking enabled action, the weight of each action will determine the probability that the given action will be selected. However, the exact selection algorithm will be determined by the specific action group. When a composed node brings together several different models with different action groups, the action groups need to be renumbered and brought together to form the action groups for the composed model, as shown in Figure 3.16. Without renumbering, there would be ambiguity between the action groups for the submodels.

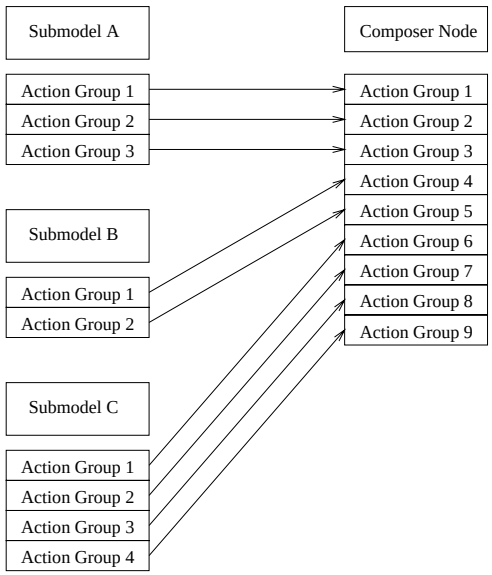


**Figure 3.16** Composition of Action Groups

**List of Shared State Variables** Each Replicate or Join contains an array of pointers to the state variables created for the sharing that it does on state variables in its submodels. A new state variable is created because the state variables shared in the submodels may have different names and types. The new state variable may have the same name as one of the shared variables, or an entirely new name. The newly created state variables will be the only state variables that can be shared by composer nodes using the current node as a submodel. This is primarily because Replicates, which apply the strong lumping theorem, cannot distinguish between their submodel replicas.

**Linked Lists of Submodels' Shared State Variables** For each newly created state variable in a composer node, we maintain a linked list of pointers to the state variables it shares in its submodels. We need to do this because of an optimization that we did in order to speed simulation, mentioned in Chapter 2. Essentially, the newly created state variable is the rep-

resentative state variable for the state variables that it shares, so any references to the shared state variables need to be redirected to the representative state variable. By doing so, when an action changes the state of the representative state variable, the simulator only needs to look for actions enabled or disabled by the representative state variable's new state, rather than look for actions affected by all of the shared state variables.

**State Size** After initialization and state sharing, each composer node will calculate its state size (including the size of the shared state) and store it for later use. Once a composer node is completely constructed, the state size will be constant, so we can use this technique to avoid doing the state size calculation again when the model is executed. The state size is calculated by adding the state sizes of the submodels (after subtracting out the size of the shared state in each of them) to the sizes of the shared state variables in the composer node.

## 3.5 Validation

Model validation refers to checking whether the model has been properly specified according to the design constraints particular to its formalism. The validation mechanism implemented in Möbius incorporates several improvements over the user interface provided in *UltraSAN*'s Replicate/Join implementation. To be specific, we need to explain what we mean when we say that an atomic model or a composer node is "validated." When a composed node is validated in Möbius, it will make the following checks:

(1) The submodels connected by the composer node still exist and are valid.

(2) All state variables shared from its submodels actually exist in those submodels.

(3) No state variable in a submodel is shared more than once.

(4) The initial states of any equivalently shared variables are identical.

With regard to validation of Replicate/Join composed models in Möbius, several features have been added to make it easier for users to correct and debug invalid composed models. Validation is done both when the user opens a composed model and when the model is saved. If there are any inconsistencies within the model, or if the model has been improperly specified, one or more dialogs will appear, informing the user about the nature and location of the problem. Some examples of these problems are enumerated below:

(1) The composed model does not have a tree-like structure with atomic model leaves and a single top node.

(2) One or more atomic models are undefined or have been deleted. An atomic model is undefined if the user has designated that an atomic model will be the submodel of a composer node, but has not yet specified which atomic model it will be.

(3) State variables with inconsistent initial states are designated as shared.

(4) More than one node has the same name. This situation is prevented in order to guarantee unambiguous reward variable specification.

(5) There is a circular dependency within the model. This situation can arise because composed models can themselves be composed.

(6) A state variable that is designated as shared is no longer present in the atomic model or composed model node that it is expected to be in.

The primary difference between composed model validation in *UltraSAN* and composed model validation in Möbius is that *UltraSAN* has less flexibility in how it allows the user to

change the structure of composed models after they have been specified. Specifically, *UltraSAN* does not allow the user to make changes to atomic models or composer nodes in a composed model without invalidating the composed model above the location of the change. After such a change, the state sharing must be redefined at each node above it. *UltraSAN* also restricts state sharing so that it can only be done on state variables with the same name.

The Möbius implementation of the Replicate/Join formalism incorporates many features to improve usability and modeling flexibility. As mentioned previously, the Möbius Replicate/Join implementation allows the user to share state variables that have different names or types, as long as the types are compatible. Additionally, we now allow the user to make changes to the atomic models or composer nodes within a composed model without automatically invalidating the model above the location of the change. However, this capability could conceivably allow the user to create an invalid model. For example, consider the following sequence:

(1) The user creates an atomic model having that has a variable $A$.

(2) The user creates a composed model sharing state variable $A$.

(3) The user deletes state variable $A$ in the atomic model.

Clearly, the composed model is now invalid, because state variable $A$ no longer exists in the atomic model, but the composed model is still specified to share state variable $A$. This indicates the need to have a validation mechanism to guarantee that composed models are structurally correct before the model is executed. Ideally, such a validation mechanism would automatically detect problems in the models that the user builds, notify the user, and correct problems if it can. To achieve these goals, we considered two different approaches to model validation:

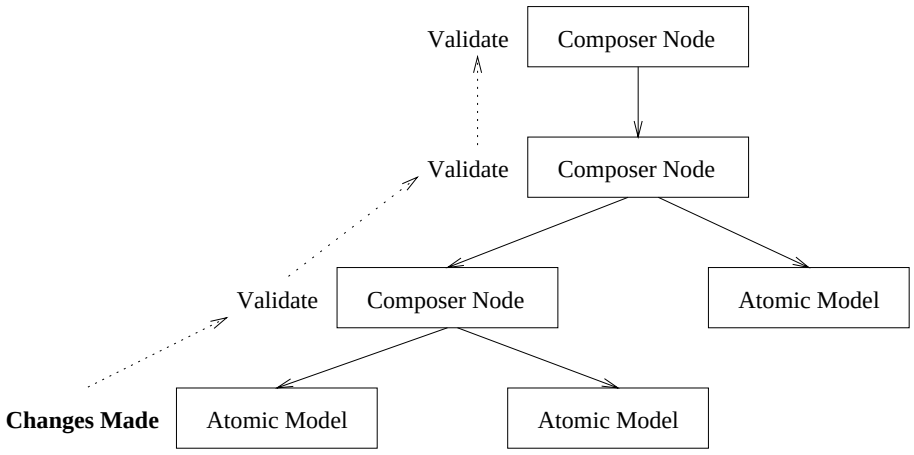


**Figure 3.17** Validation Approach 1

(1) When a change is made to an atomic model or to a composed node within a composed model, immediately propagate information regarding the change up to higher nodes and validate them in order.

(2) When a composed node is edited, validate composer nodes and atomic models beneath it, from the submodels up to (and including) itself.

The two approaches to validation are depicted in Figures 3.17 and 3.18.

The first approach to validation was implemented initially, because we believed that it would give the user immediate feedback if changes were made that invalidated parts of any composed models. However, this method turned out to be extremely time-consuming, because any time a small change was made in the lower part of a large model, the entire model would need to be validated; this resulted in inefficiency when several small changes to different atomic models needed to be made. For example, if all of the atomic models needed to be changed, the composed models, performance variables, and studies dependent on them would need to be validated over and over. Instead of doing the validation for the composed models, performance

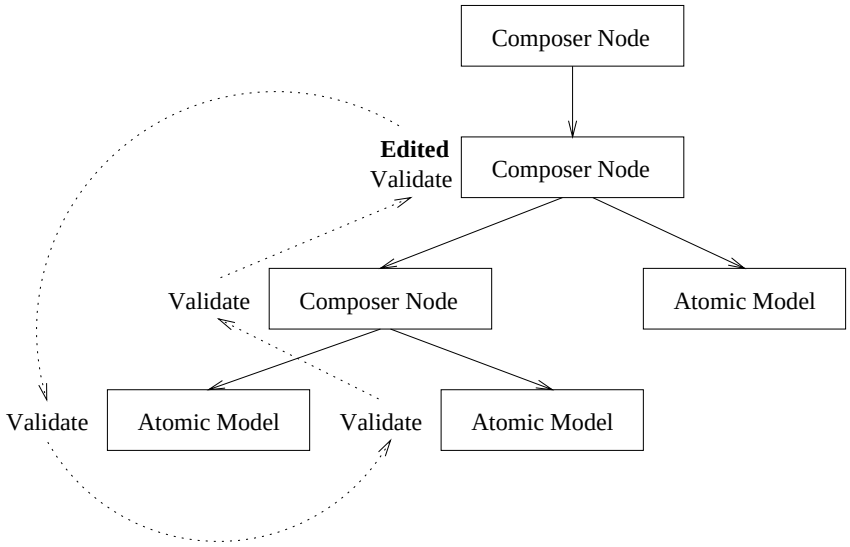


**Figure 3.18** Validation Approach 2

variables, and studies repeatedly for each atomic model changed, the validation should only need to be done at the end of the changes.

Another significant problem with this approach to validation is that it becomes inconvenient for the user to change the initial state of state variables that are shared in composed models. Consider the following scenario:

(1) The user creates two atomic models in the stochastic Petri net formalism, each having a Place $A$ with initial marking 5.

(2) The user creates a composed model that shares the two Places $A$ in the atomic models.

(3) The user decides that an initial marking of 3 would be more appropriate for Place $A$.

(4) The user attempts to change the initial marking in one of the atomic models to 3, but it has already been shared with another Place having marking 5. The change, if allowed, will immediately propagate up to the composed model. The composed model will be

invalidated unnecessarily because the user does not have a chance to change the other atomic model before the change propagates.

From this scenario, it becomes evident that immediate propagation and validation of changes can make it difficult for the user to make even minor changes to parts of composed models. Fortunately, these difficulties can be avoided by using the second approach to validation, which is the approach that is used in the current implementation of Möbius.

With the second approach to validation, when a composed model needs to be validated, all of its child models (whether atomic or composed) are first validated via a post-order traversal, as shown in Figure 3.18. In order to reduce the overhead of doing the traversal and validation of all of a composed model's children, a valid bit is associated with each atomic or composed model. As soon as a model is confirmed to be valid, the valid bit is set until the model is next modified. When validation on a model is requested, the valid bit will be queried first; if the bit is set, then the model can be assumed to be valid, and does not need to be loaded and checked thoroughly. For more information about validation mechanisms in Möbius, see Appendix A.

# CHAPTER 4

# GRAPH COMPOSER FORMALISM

## 4.1 Introduction

In this chapter, we will describe the second composer formalism which was integrated into the Möbius modeling framework. The initial composer formalism to be implemented in Möbius was the Replicate/Join formalism, also implemented in *UltraSAN*. With the graph composer formalism, we demonstrate that new composer formalisms may be integrated smoothly into the Möbius environment through exploitation of the generic state-sharing framework that we have established.

The main motivation behind the design of the graph formalism was the desire to create composed models with arbitrary node configurations. With the previously discussed Replicate/Join formalism, composed models were limited to tree-like, hierarchical structures. While this is sufficient and even convenient for the representation of many systems, it is difficult to use the Replicate/Join formalism to build composed models representing other types of systems, such as those with ring-like structures (like those seen in networks).

Our graph composer formalism uses only two types of nodes to build composed models: Joins and submodels. A Join may only be connected with one or more submodels, and submodels may be connected with one or more Joins. Joins may not be connected to Joins, and submodels may not be connected directly to submodels. Submodels may be atomic models or composed

models. Joins are used to connect submodels by sharing portions of their state, otherwise known as state variables. Each state variable in a submodel may only be shared at a single Join.

The graph composer formalism allows us to build and study models that have no convenient representation in the Replicate/Join composer formalism. In the following section, we will demonstrate this by showing how we can represent the classic dining philosophers problem with a composed model in the graph composer formalism. We then discuss implementation details, including an overview of the C++ classes used in the implementation. Finally, we explain model validition for graph formalism composed models.

## 4.2 Dining Philosophers Problem

At the center of a round table in a faraway boarding house, there is a large bowl of steaming rice. Around this table are five place settings, and behind each place setting sits a philosopher. Five chopsticks, one between each pair of place settings, are the only utensils at the table.

The five philosophers lead a simple life: they spend their time thinking and eating. Of course, a philosopher who wishes to eat must first have in hand two chopsticks, and the etiquette of the boarding house requires that the hungry philosopher pick up his chopsticks in a seemly manner, one at a time. Indeed, rude behavior of any sort is frowned upon: there can be no grabbing of two chopsticks at once, no breaking a single chopstick in two, no passing of chopsticks around the table, and no eating with fingers.

Several things are clear about this peculiar lifestyle. At any given moment, no more than two of the five philosophers may eat. The arrangement of the chopsticks around the table ensures that when one philosopher is eating, neither of the philosophers seated next to him can pick up enough chopsticks to begin eating.

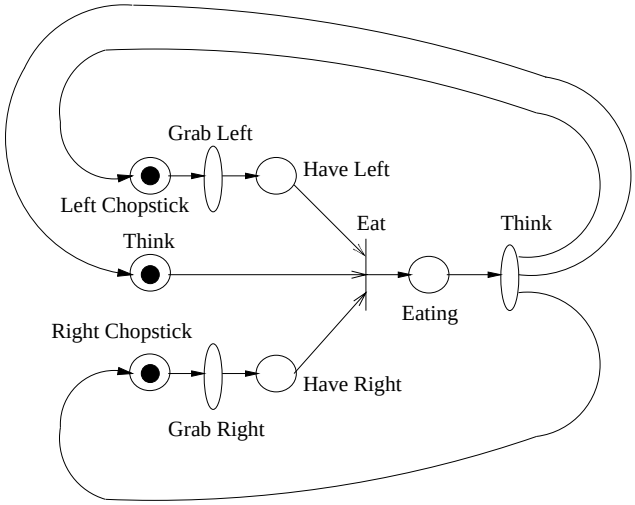


**Figure 4.1** Atomic Model Depicting One Dining Philosopher

Using the graph composer formalism, it is easy for us to construct a model of the dining philosophers problem. We begin with the Stochastic Petri Net atomic model shown in Figure 4.1. Each philosopher begins in the thinking state with a chopstick both to his right and to his left. In order for the philosopher to eat, he needs to have both chopsticks available. When the philosopher is eating, neither chopstick that he is using will be available for another philosopher to use. When he finishes eating, he replaces both chopsticks and returns to his thinking.

Construction of the composed model with the graph composer formalism is straightforward. We begin by placing the desired number of philosophers in a ring, and placing join nodes between adjacent philosophers. At the join nodes, we share each philosopher's right chopstick with his right neighbor's left chopstick, and vice versa. The final composed model is shown in Figure 4.2.

The main advantage of the graph composer formalism is that it allows the modeler to partition a large, complicated atomic model in an arbitrary fashion, and then combine the pieces to

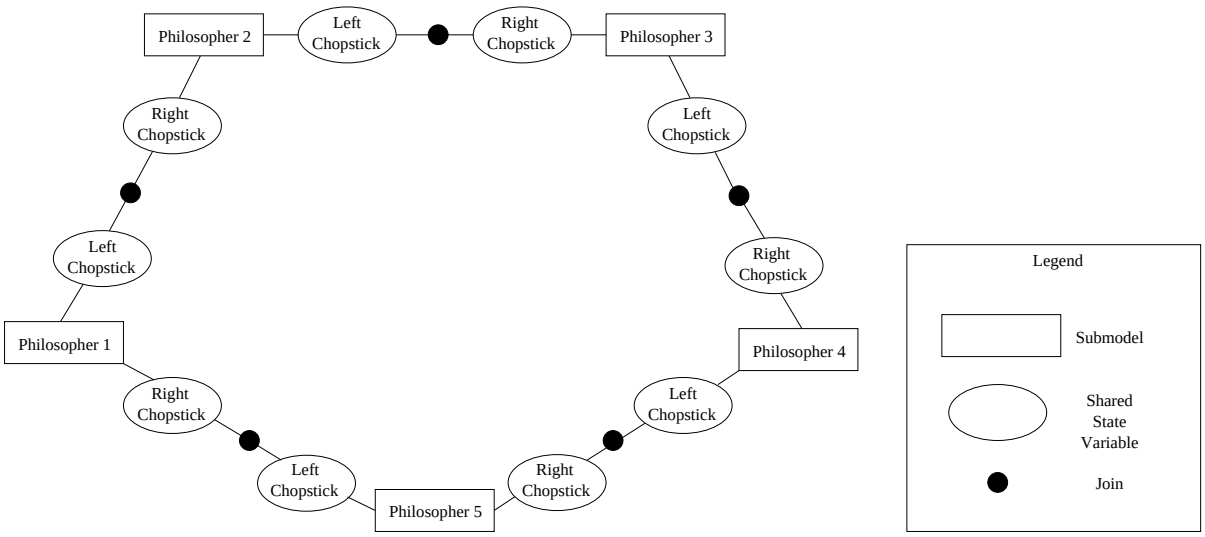


**Figure 4.2** Composed Model for Dining Philosopher Problem

form a unified composed model that can be treated as an atomic model. This strength becomes more and more beneficial as models become larger and larger, since it can be cumbersome for the user to work with extremely large atomic models. Instead, the user can organize the model into modules and combine them into a graph formalism composed model.

## 4.3 Implementation

In this section, we will discuss the implementation details of the graph formalism within the Möbius modeling framework. The discussion will cover the following items:

- Overview of the differences between Replicate/Join formalism composed models and graph formalism composed models.
- Representation of state.
- Integration of the graph composer into the Möbius modeling environment.

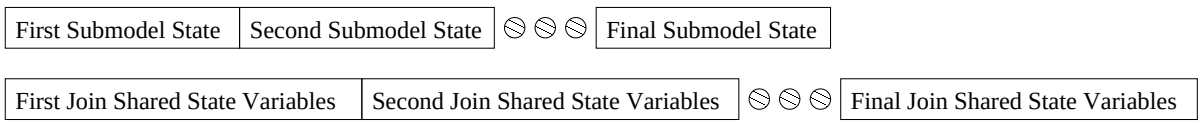


**Figure 4.3** State Representation for Graph Formalism Composed Models

### 4.3.1 Differences between Replicate/Join and graph formalisms

The implementation of the composed model editor for the graph composer bears many similarities to the implementation of the Replicate/Join composer formalism. For example, graph formalism composed models maintain a consistent external representation of state, with a representation very similar to that of Joins. However, the structure of graph formalism composed models is very different from that of Replicate/Join composed models. Replicate/Join composed models must conform to a tree-like structure that can have arbitrary depth, while graph formalism composed models always have a depth of two, where the top level of the model consists entirely of Joins and the bottom level of the model consists entirely of submodels. This fundamental difference gives rise to the implementation specifics that we will explore in this section, and to the differing validation methodology that we will explain in the next section.

### 4.3.2 Representation of state

As mentioned previously, the state representation for graph formalism composed models is similar to that of Joins in the Replicate/Join composer formalism. The state of a graph composed model is the sum of the states of its atomic models. When state variables are shared across atomic models and the state of the composed model is requested, we only need to report the state of the newly created shared variable, which will reside in the Join node that is the parent of the applicable atomic models. This approach leads to the representation of state shown in Figure 4.3.

### 4.3.3 Integrating the graph composer into Möbius

In order to comply with the requirements of composed models within the Möbius modeling framework, it must be possible to compose graph formalism composed models with other atomic or composed models to form new composed models. This means that it must be possible to treat graph formalism composed models as atomic models. With Replicate/Join composed models, this requirement is almost trivial because of their tree-like structure; we can simply take the top node of the tree and treat that like an atomic model. However, it is more difficult with graph formalism composed models because we are not guaranteed to have a distinct top-level node. Therefore, we need a way to represent the structure and behavior of the composed model in such a way that it can be treated as an atomic model.

To allow graph formalism composed models to be treated as atomic models, we create a logical top-level node, which we will refer to as a *wrapper*. One wrapper object will be associated with every graph formalism composed model. Essentially, the wrapper node acts as an intermediary between other parts of the tool and the joins and submodels in the composed model. The wrapper will accept function calls from other models and/or solvers, and then determine the appropriate way to pass on the call to the Joins and submodels within the composed model. For example, when another model requests the list of state variables from a graph formalism composed model, the wrapper will query the Joins and submodels in succession to obtain the list of state variables for the entire model; when the list is complete, the wrapper will return it to the requester. Another example is the case where another model requests the list of actions from a graph formalism composed model; in this case, the wrapper will accumulate the list of actions only from the submodels, neglecting the Joins because the actions for the Join

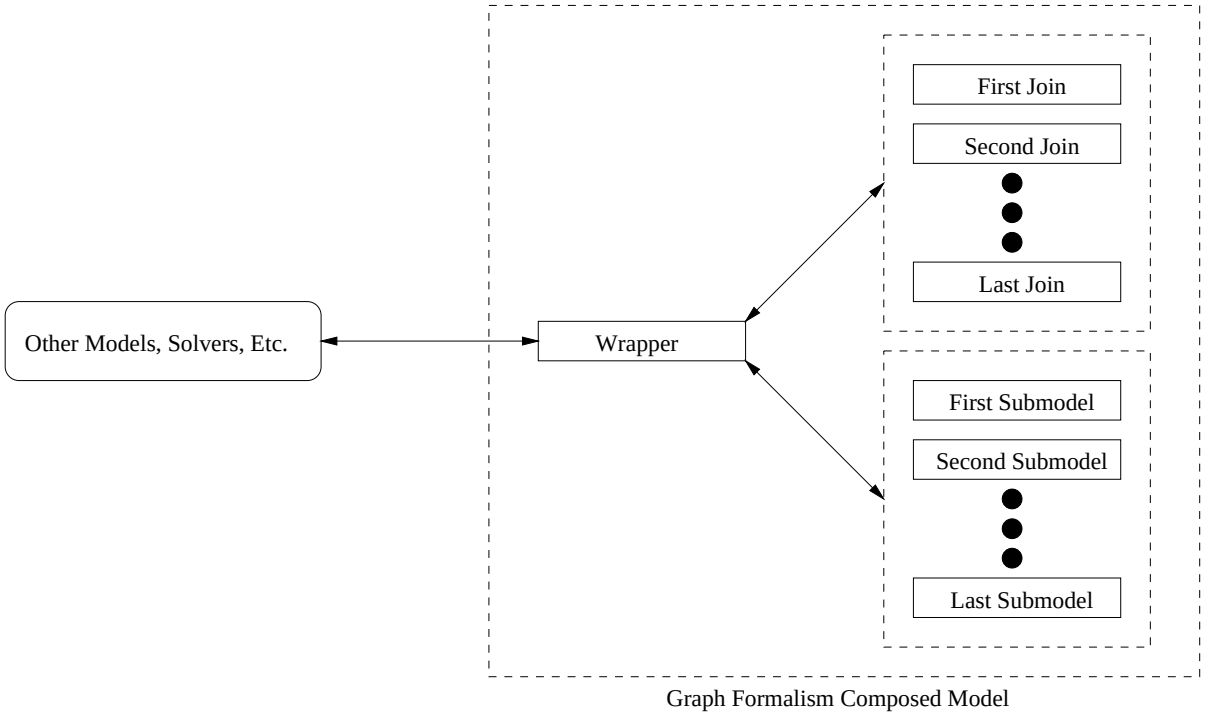


**Figure 4.4** Graph Formalism Composed Models within Möbius

nodes will already be included in the actions for the submodels. A block diagram depicting the role of the wrapper is given in Figure 4.4.

### 4.3.4 C++ classes

Our C++ implementation of the graph composer formalism consists of three main classes: BaseGraphComposerClass, Join, and Wrapper. BaseGraphComposerClass is the base class from which Join and Wrapper are derived, containing functionality and initialization routines common to the two derived classes.

**BaseGraphComposerClass** BaseGraphComposerClass is derived from BaseModelClass, the class from which all atomic models are also derived. BaseGraphComposerClass contains an array of pointers to the submodels in the current composed model. The BaseGraphComposerClass

constructor handles the parts of composer node initialization that are common to Joins and Wrappers; for example, it sets up the actions and groups. The role of BaseGraphComposerClass is very similar to the role of BaseComposerClass from the Replicate/Join formalism.

**Join** For all practical purposes, Joins in the graph composer formalism are identical in structure and behavior to Joins from the Replicate/Join formalism. The main difference is that in a given graph formalism composed model, Joins can only have submodels that are either atomic models or other composed models. Additionally, the user is not required to share a state variable from a submodel in a Join in order to be able to share it later in a higher-level composed model.

**Wrapper** As mentioned previously, wrappers serve as intermediaries between other models or solvers and the Joins and submodels within a graph formalism composed model. Wrappers will contain an array of pointers to the Joins in the model, in addition to the array of pointers to submodels inherited from BaseGraphComposerClass. For each graph formalism composed model, a wrapper will automatically be instantiated and initialized, effectively making the wrapper a logical composer node that communicates the structure and behavior of its underlying Joins and submodels.

## 4.4 Validation

Validation of graph formalism composed models is almost identical to that of Replicate/Join composed models. In particular, the general rules for determining whether a composed model is valid are exactly the same:

(1) The submodels connected by the composer node still exist and are valid.

(2) All state variables shared from its submodels actually exist in those submodels.

(3) No state variable in a submodel is shared more than once.

(4) The initial states of any equivalently shared variables are identical.

The primary difference in the validation algorithm for graph formalism composed models is that we do not do a post-order traversal of a tree-like structure of composed nodes. Instead, we simply make the following sequence of checks to validate the structural integrity of graph formalism composed models:

(1) All submodels are connected to at least one Join node.

(2) All Joins are connected to at least two submodels.

(3) No Joins are connected to other Joins.

(4) No submodels are connected to other submodels directly.

# CHAPTER 5

# CONCLUSION AND FUTURE WORK

In this thesis, we have discussed model composition within the Möbius modeling environment. First, we have presented the state sharing framework that we have established as a generic means of constructing composed models from atomic models. Then, we have shown two implementations of composer formalisms, the Replicate/Join and graph formalisms, on top of the state-sharing framework. These two composer formalism implementations demonstrate how new composer formalism interfaces can be integrated seamlessly into the Möbius tool, allowing model construction, simulation, and solution as soon as the new interface is added.

Möbius was designed to be an extensible modeling environment, meaning that there are numerous opportunities to enhance the capabilities of the tool. The most obvious way would be to design and implement new formalisms customized to meet specific modeling needs. More work also can be done to improve the appearance and usability of the Möbius graphical user interface. As modern technology continues to advance at an aggressive rate, it is possible that the base classes may need to be extended to adapt to the more demanding modeling needs of future technologies.

The content of this thesis is focused on model composition within Möbius, so we will suggest a specific project related to a new composer formalism. The new composer formalism will extend the existing graph composer to take advantage of symmetries within the model. Sophisticated mathematical techniques useful for symmetric state space reduction were developed by W. D. Obal II and presented in his Ph.D. dissertation [12]. Implementing these techniques within a

Möbius composer editor could very well yield larger reductions in model state spaces than those afforded by the Replicate/Join composer formalism. For more details, see [12].

The current implementation of Möbius supports all of the functionality of *UltraSAN* and also includes several enhancements to usability that we have designed. The key strength of Möbius, however, lies in its extensibility, as evidenced by the addition of the graph composer formalism. It is our hope that the Möbius state-sharing framework that we have established will provide a powerful and flexible means to address the modeling needs of future engineers in industry and academia.

# APPENDIX A

# Project Manager

The Möbius project manager allows the user to organize related atomic models, composed models, performance variables, studies, solver parameters, simulation parameters, state spaces, results, and other relevant information in the form of projects. Using projects to organize information greatly enhances usability, because it gives the user a convenient way to view and manipulate only the data that he or she is interested in. Projects also constitute an essential part of interface-to-interface interactions by providing the backbone for communication between them.

## A.1 Project Manager Window

When the user runs the Möbius application from the command line, the first window that will appear is that of the Möbius Project Manager. The project manager is the focal point of the application, from which projects may be manipulated and application settings may be tuned. Additionally, the user may choose to install or remove different formalism interfaces.

Project manipulation is accomplished via the pull-down Project Menu or by the use of hot keys. Following are the different project functions that are currently supported:

(1) Create and name a new project.

(2) Open and edit an existing project.

(3) Delete an existing project.

(4) Rename an existing project.

(5) Copy an existing project, giving the copy a new name.

(6) Archive an existing project into a tar'd, gzip'd bundle.

The user may wish to customize his or her application settings, which will apply to all projects. The project manager application parameters are divided into two categories: general application settings and compiler settings. General application settings include the following:

- Mobius Installation Path - Where the user has installed the Möbius application, meaning the C++ archives, Java classes, and other files needed to run Möbius. This path must be specified correctly in order for the functional C++ models to compile, link, and execute.
- *UltraSAN* Root Project Directory - The location of the *UltraSAN* root project directory, if any. This is useful if the user has existing projects that were created using *UltraSAN* and wants to import them into Möbius.
- Root Project Directory - Where the user wishes to save and access projects created in Möbius.
- Interface Cache Size - As a speed optimization, a certain number of models will be saved in memory for each open project. In order to control memory consumption, the user may specify the number of models that should be cached.
- Text Component Height and Width Scaling Factors - Because Java graphical user interface (GUI) components appear differently on different platforms, the user may customize the size of the Java GUI components that Möbius displays.

The second category of application settings is that of compiler settings. Möbius currently supports model development and simulation across multiple platforms. The platforms that presently are supported in Möbius are Solaris, HP-UX, and Linux. Möbius allows the user to specify separate compilation parameters for each supported platform. These options include the following:

- Compiler - The full path and executable name of the C++ compiler to be used for model compilation.
- Compiler Options - Option flags to be used in conjunction with the specified compiler.
- Linker - The full path and executable name of the linker to link the C++ object files or archives.
- Linker Options - Option flags to be used in conjunction with the specified linker.
- Make Command - The full path and executable name of the make utility.
- Archive Command - The full path and executable name of the archive utility needed to combine object files into archives.
- Ranlib Command - The full path and executable name of the ranlib utility used to regenerate the symbol tables for the C++ archives created.

The user may also wish to install or remove formalism interfaces for a number of reasons. When new formalisms are developed and become available, the user may wish to make use of them in order to increase the tool's capabilities for modeling and solution. Alternatively, the user may wish to disable unneeded functionality to reduce the amount of time needed to load

the tool. Windows that allow the user to install or disable formalism interfaces can be accessed via the project manager's pull-down menus.

The project manager serves all projects as a central source of information about the platform and computer system that Möbius is running on. This information is needed in order for platform-specific executable code to be stored in the right location so that it can be linked in later. Other information that the project manager will determine and distribute dynamically includes the location of the user's home directory, the user's application settings, and the system-dependent file separator string. The consumers of this information are the projects and formalism interfaces.

## A.2 Projects

A project can be viewed as a collection of related atomic and composed models, performance measure specifications, simulation and solution parameters, and results. By opening a project window, the user can make changes to these and/or run simulations, generate state spaces, and perform mathematical solution. Users open projects from the main project manager window. Within each project window, several tabs list the different categories of items that may be manipulated. When a tab is selected, the current items in the corresponding category are displayed in a list box. After selecting an item, the user may initiate an action on the item by clicking on the buttons to the right of the item list. Several existing categories and possible actions are listed below:

- Atomic Models - Create, edit, delete, or import an atomic model.
- Composed Models - Create, edit, or delete a composed model.

- Performance Variables - Create, edit, import, or delete a set of performance variables.
- Studies - Create, edit, import, or delete a study.
- Solvers - Run a simulation or generate a state space.
- Analytic Solvers - Run an analytic solver on a previously generated state space.

Möbius allows the user to open several projects at the same time. However, there are restrictions on how many items may be edited simultaneously in an open project. Any number of atomic models may be edited at the same time, but we currently restrict the user to having only one composed model open at a given time. Additionally, model editors of different categories (e.g., atomic and composed) may not be open at the same time. The reason for these restrictions is to prevent the user from making a change that will put the project in an inconsistent state. Consider the following situation:

(1) The user creates an atomic model with state variables $A$, $B$, and $C$.

(2) The user saves the atomic model and closes the atomic model editor.

(3) The user creates a composed model that shares state variables $A$ and $B$, but does not save the model yet.

(4) While the composed model editor is still open, the user opens the atomic model editor.

(5) The user deletes state variable $A$.

(6) The user renames state variable $C$ to be state variable $A$ and saves the atomic model.

(7) The user saves the composed model.

Clearly, the sequence of events creates an ambiguity in the composed model. The composed model is specified as sharing state variables $A$ and $B$, but it is difficult to say whether it should share the original state variable $A$ or the renamed state variable $C$. In order to resolve situations like this, we would need to establish a set of semantics to dictate the behavior of the tool. It would be preferable to allow on-the-fly updates to models, but this would greatly increase the complexity of the implementation and could result in situations in which the tool behaves in ways unexpected or undesired by the user. For these reasons, we decided to enforce the simple restrictions of being able to open only one composed model at a time and being able to open items from only one category at a time. These restrictions ensure that the user will modify the project in an ordered, unambiguous fashion and simplify the complexity of the implementation considerably.

Projects serve as the backbone for communication between formalism interfaces. From an object-oriented standpoint, it is undesirable for an interface to communicate directly with or record information about other interfaces. For example, consider what would happen if a composed model recorded the exact directory paths and file names for the atomic models that it includes, and relied on those directory paths and file names to obtain information about the atomic models. Should an atomic model's files be renamed or moved to a different directory, the composed model would be unable to locate the files needed to obtain information about its atomic models.

In order to simplify each formalism interface's interactions with other formalism interfaces, we require that each created model have a name unique to the project that contains it. For each created model, the project will maintain the necessary information regarding the model, including directory paths and file names for its files, and whether or not the model has been

changed. When a model is changed, the project will mandate that any models that depend upon it be validated and have their object files recompiled.

A simple mechanism is provided to allow the user a way of specifying interactions with other models. When another model needs to be selected (for example, in building a composed model, the user will need to specify which atomic models will be included), a simple call is made to the current project to bring up a child selection dialog box. The project will determine which other models are valid candidates for interaction with the calling model, and display their names in the dialog box. After the user selects a model from the dialog box, its name will be returned to the calling model. The calling model will then store the name for future use. When a model requires specific information about another model, it can simply communicate with its containing project, passing it the other model's name.

# REFERENCES


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

[2] A. Movaghar and J. F. Meyer, "Performability modeling with stochastic activity networks," in *Proc. 1984 Real-Time Systems Symposium*, (Austin, TX), pp. 215–224, Dec. 1984.

[3] D. Daly, D. D. Deavours, J. M. Doyle, A. J. Stillman, and P. G. Webster, "Möbius: An extensible tool for performance and dependability modeling," in *Digest of Fast Abstracts, 1999 International Symposium on Fault-Tolerant Computing*, (Madison, WI), pp. 15–16, June 1999.

[4] J. M. Doyle, "Abstract model specification using the Möbius modeling tool," Master's thesis, Dept. of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1999.

[5] G. P. Kavanaugh III, "Design and implementation of an extensible tool for performance and dependability model evaluation," Master's thesis, Dept. of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1998.

[6] J. M. Sowder, "State-space generation techniques in the Möbius modeling framework," Master's thesis, Dept. of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1998.

[7] A. L. Williamson, "Discrete event simulation in the Möbius modeling framework," Master's thesis, Dept. of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1998.

[8] W. H. Sanders, W. D. Obal II, M. A. Qureshi, and F. K. Widjanarko, "The *UltraSAN* modeling environment," *Performance Evaluation*, vol. 24, pp. 89–115, 1995.

[9] W. H. Sanders and J. F. Meyer, "Reduced base model construction methods for stochastic activity networks," *IEEE Journal on Selected Areas in Communications*, vol. 9, pp. 25–36, Jan. 1991.

[10] W. D. Obal II and W. H. Sanders, "Importance sampling simulation in *UltraSAN*," *Simulation*, vol. 62, pp. 98–111, Feb. 1994.

[11] J. C. Kemeny and J. L. Snell, *Finite Markov Chains*. Princeton, NJ: D. Van Nostrand Co., Inc., 1969.

[12] W. D. Obal II, "Measure-adaptive state-space construction methods," PhD dissertation, Dept. of Electrical and Computer Engineering, University of Arizona, Tucson, AZ, 1998.