ABSTRACT MODEL SPECIFICATION USING THE MÖBIUS MODELING TOOL

BY

JAY M. DOYLE

B.S., Syracuse University, 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, 2000

Urbana, Illinois

ABSTRACT

Predicting the performance and dependability of modern computer and communication systems has become increasingly difficult due to the complexity of such systems. In order to obtain accurate measures of a system's performance and dependability, it is often necessary to have detailed models of the system's components, which vary in nature (e.g., hardware, software, networks, and operating system). Therefore, in order to model modern computer and communication systems accurately, tools are needed to model each subsystem to an appropriate level of detail. This can be done by building a tool that allows a modeler to specify different parts of a system in different modeling formalisms. Modelers could then develop and use the modeling formalisms that best suit each particular subsystem. Such a modeling tool would also need to support ways of composing these subsystem specifications into a single model of the system and solving the model for the desired measures.

This thesis proposes a software framework for a modeling tool that supports model specification using a variety of modeling formalisms. This proposed software framework allows models expressed in different modeling formalisms to communicate with other models and solvers through an abstract functional interface. This thesis will define this abstract functional interface and show how it can be used to support multiple modeling formalisms. This thesis also shows how a high-level stochastic modeling formalism (stochastic activity networks) can be implemented in the proposed software framework.

To my wife Heather, for all the lonely nights alone.

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor William H. Sanders, for technical advice and support on the Möbius project. I would also like to thank all the members of the Möbius group, both past and present, who made this tool a reality: Amy Christensen, David Daly, Dan Deavours, G. P. Kavanaugh, Doug Obal, John Sowder, Aaron Stillman, Alex Williamson, and Patrick Webster.

I would also like to thank the Defense Advanced Research Project Agency, Information Technology Office, for funding under contract DABT63-96-C-0069, and the National Science Foundation for funding under the Next Generation Software Program.

TABLE OF CONTENTS

CHAPTER PAGE

1 INTRODUCTION 1
- 1.1 Modeling Systems for Performance and Dependability 1
- 1.2 Motivation 2
- 1.3 Introduction to Möbius Concepts 4
- 1.4 Research Objectives 5

2 THE MÖBIUS ABSTRACT MODEL SPECIFICATION 7
- 2.1 Introduction 7
- 2.2 Definition of Symbols 9
- 2.3 Overview of the Möbius Framework 10
- 2.4 Möbius Abstract Functional Interface 13
  - 2.4.1 State variables in the Möbius abstract functional interface 13
  - 2.4.2 Actions in the Möbius abstract functional interface 18
  - 2.4.3 Groups in the Möbius abstract functional interface 26
  - 2.4.4 Models in the Möbius abstract functional interface 30
- 2.5 Execution Policy 32

3 IMPLEMENTING THE ABSTRACT FUNCTIONAL INTERFACE 36
- 3.1 `BaseStateVariableClass` 38
  - 3.1.1 Sharing through state equivalence 41
  - 3.1.2 Sharing through functions of state 44
  - 3.1.3 Implementation of the `SharableSV` class 47
  - 3.1.4 Unidirectional functionally sharable state variables 50
- 3.2 `BaseActionClass` 52
- 3.3 `BaseGroupClass` 61
- 3.4 `BaseModelClass` 64
- 3.5 Additional Data Structures 69
  - 3.5.1 `List` 70
  - 3.5.2 `UserDistributions` 71
- 3.6 Additional Functions 72

4 REALIZATION OF AN ATOMIC MODEL FORMALISM THROUGH THE ABSTRACT FUNCTIONAL INTERFACE 73
- 4.1 Introduction to Stochastic Activity Networks Theory 73
- 4.2 Stochastic Activity Networks Implementation 75

4.3 Realization of SANs Through the Abstract Functional Interface 76
4.3.1 Places as state variables 76
4.3.2 Activities as actions 77
4.3.3 Multi-case activities as postselection groups 84
4.3.4 SAN models as Möbius models 85

5 CONCLUSIONS AND FUTURE RESEARCH 86

REFERENCES 88

LIST OF TABLES

Table Page

2.1 Definition of symbols . . . 9
2.2 Rules for constructing all state variable types in the Möbius framework . . . 10
2.3 Action functions defined by the Möbius framework . . . 12
2.4 The set of functions and sets defined on groups in the Möbius framework . . . 13
2.5 Definition of basic types in the abstract functional interface . . . 14
2.6 An example state variable . . . 16
2.7 Functions defined on actions . . . 18
2.8 Action attributes . . . 18
2.9 Possible outcomes of evaluating an action's **ReactivationFunction** . . . 22
2.10 A list of model functions . . . 31

3.1 Methods defined on `BaseStateVariableClass` . . . 39
3.2 Methods defined on `SharableSV` class . . . 49
3.3 `SharableSV` data members . . . 50
3.4 Methods defined on `ReadOnlySV` template class . . . 51
3.5 Methods defined on `BaseActionClass` that implement abstract functional interface action functions . . . 54
3.6 `BaseActionClass` methods used to access information about `AffectedStateVariables` and `EnablingStateVariables` data structures . . . 57
3.7 `BaseActionClass` attributes data structures . . . 58
3.8 Supported distribution functions in `UserDistributions` class . . . 59
3.9 `BaseActionClass` data structures used by simulators . . . 59
3.10 Action state data structures . . . 60
3.11 Performance variable action data structures . . . 62
3.12 Methods defined on `BaseGroupClass` . . . 63
3.13 Methods defined on `BaseModelClass` . . . 66
3.14 Data structures defined on `BaseModelClass` . . . 69
3.15 Methods defined on `List<class C>` template class . . . 71
3.16 A list of template functions for reading and writing across word boundaries . . . 72

4.1 Syntax differences between *UltraSAN* and Möbius . . . 77
4.2 SAN sources for action method realizations . . . 78

LIST OF FIGURES

Figure Page

3.1 Initialization algorithm for state variable `AffectingActions` objects 42
3.2 Example of two state variables not sharing state 44
3.3 Example of two state variables sharing the same state 45
3.4 Example of two previously shared state variables of order 2 joined to form a single shared state variable of order 4 46
3.5 Class hierarchy for state variable classes 48
3.6 Three different formalisms with functionally-sharable state variables 53
3.7 An example of equivalence and functional state sharing 55
3.8 Algorithm for selecting a group member 65
3.9 Algorithm for calculation of member distribution function 65

4.1 Implementations of the `SampleDistribution` method for SAN activities . . . 79
4.2 Sample implementation of the `ReturnDistributionParameters` method for SAN activities 80
4.3 Sample implementation of the `Enabled` method for SAN activities 81
4.4 Two sample SAN activity definitions 82
4.5 Sample implementations of the `Fire` method for SAN activities 83

CHAPTER 1

INTRODUCTION

1.1 Modeling Systems for Performance and Dependability

Over the past decade, many advances in software tools used for modeling the performance and dependability of computer systems have made it possible for engineers to model large-scale systems with a reasonable amount of computer resources. Nevertheless, the complexity of the systems being modeled by these tools has also increased to such a level that the salient features of a system often cannot be accurately represented in a single model. For example, modern computer system designs often contain many distinct parts and levels of complexity (e.g., operating system, hardware, computer network, and software). Each of these model parts can have an important impact on both performance and dependability. Thus, to model large systems to the appropriate level of detail, we will need software tools that enable a modeler to represent each system component accurately with a reasonable amount of detail.

Different tools approach model specification in different ways. For instance, tools used for performance analysis require inputs that include information about system throughput, latencies, and competition for system resources, whereas modeling tools geared toward dependability would most likely require the modeler to define system components in terms of failure rates. Thus we see that there are many different ways to approach model specification. We often use the term *modeling formalisms* to refer to these different approaches to model specification. A *modeling formalism* is thus a language for expressing a model [1]. Such languages contain rules and constructs that define how models expressed in the formalism operate.

1.2 Motivation

Sanders [2] classifies current stochastic modeling tools into several broad categories: single-formalism tools, software environments that incorporate multiple tools, and integrated modeling environments. Many single-formalism tools use either a queuing network or stochastic Petri net formalism. Some of the more popular tools based upon queuing networks include DyQN-Tool [3], LQNS [4], QNAP2 [5], RESQ [6], and RESQME [7]. Tools based upon queuing networks are usually geared towards building performance models. Many extensions have been added to queuing networks to make them more versatile for modeling systems with complex interactions. There are also many tools based upon stochastic Petri nets; see [8] for a comprehensive listing. One of the most important of these tools for our own research has been *UltraSAN* [9], which is based upon stochastic activity networks.

Modeling tools in the second category, software environments, aim to combine two or more pre-existing modeling tools into a single, cohesive environment. One can realize such an environment through a common user interface with which a user can easily move from one modeling tool to another. In essence, each of these software environments provides a modeling toolbox. This approach is mentioned by Smith [10]; some implementations include IMSE (integrated modeling support environment) [11], IDEAS (integrated design environment for assessment of computer systems and communication networks) [12], and Freud [13].

The last category of modeling tools aims to build large models by designing a framework that can support multiple formalisms and a variety of model solution techniques. One way to implement this design philosophy is to connect models in different formalisms by exchanging results. Two current tools, SHARPE [14] and SMART [15], both use this approach to build models using two or more modeling formalisms. Another approach in this regard is to convert models into a single universal modeling language. This is the approach taken by DEDS (Discrete Event Dynamic System) [16], which uses an abstract Petri net notation as its universal modeling language. Lastly, one may build a framework in which state, event, or results can be shared among models expressed in different modeling formalisms. This is the approach used by Möbius [17, 1, 18].

The motivation for building the Möbius tool is the observation that no single stochastic, discrete-event formalism has shown itself to be the best for building and solving models across different application domains. Different application domains often have different emphases (e.g., performance, dependability, reliability, and validation). In the past, different modeling formalisms have been developed to address the needs of specific application domains. Such a proliferation of modeling formalisms is an indication that even within a specific application domain, there may not be a single best modeling formalism for developing performance and reliability models.

Determining the relative merits of one formalism over another is a subjective process. One may ask the following questions to judge the merits of a particular formalism for a specific model or application domain:

- Can the formalism accurately model the state of the system? If abstractions are necessary to represent the system, do these abstractions oversimplify the representation, thus making it useless?
- Can the formalism accurately model how a system changes state?
- How complicated is the model specification process?
- Given that a model is specified in a particular formalism, what types of solution methods are available?
- What types of reward structures can the modeler define? What metrics can we solve for?
- What does the modeler find the most familiar or comfortable?

Just as we concluded that there is no single modeling formalism appropriate for all applications, we also believe that there is no single solution method that is best for all models. Simulation-based solution techniques allow for broader model specification, but are less likely to be able to capture the effects of rare but important events. Also, solution via simulation often requires a great deal of time to ensure confidence in the model's solutions. Analytical/numerical-based techniques provide exact solutions for reward metrics, but have

many model specification restrictions. For instance, many analytical/numerical solution methods are based on solving Markov or semi-Markov processes. A Markov process representation requires that state changes be exponentially distributed. In many cases, that requirement may not reflect the true operation of the modeled system's operation; even if it does, the size of the model's state space may make analytical solution infeasible.

Given that there appears to be neither a single model formalism best for all application domains nor a single, efficient, and appropriate solution method for all application domains and formalisms, we believe that the best framework for building modeling tools is one in which many different modeling formalisms and solution methods can be easily integrated. This framework should allow for rapid integration of both new modeling formalisms and model solutions. New techniques in model specification and solution are often hindered by the need to build a complete tool in order to realize a novel concept. Ideally, model specification and model solution can be done in a more independent fashion. Furthermore, a modeling environment that allows parts of a model to be specified in different modeling formalisms would be advantageous, since large, complex systems are often spread over several different application domains; a modeler can choose the best formalism for each part of a model and later combine them together to form one large, heterogeneous model.

1.3 Introduction to Möbius Concepts

The Möbius framework [19] defines important concepts that we are using to build the Möbius tool. One of the important contributions of the Möbius framework is its attempt to classify types of modeling formalisms and models. These new terms help to define the way we approach and think about model building in the Möbius tool.

At the most basic level, the Möbius tool defines *atomic models* and *atomic model formalisms*. *Atomic model formalisms* are model formalisms for describing atomic models. *Atomic models* are self-contained (but not always complete) models, each of which is expressed entirely in a single formalism. Atomic models often encapsulate the functionality of a small part of a large system. Thus atomic models are the building blocks for larger models.

Since large atomic models can become unwieldy and complex, the Möbius framework supports the concept of "composed model formalisms." *Composed model formalisms* are modeling formalisms that specify how one or more models can be structurally joined to form a single, larger model. The result of composing models together in a composed model formalism is a new model that we refer to as a *composed model*. Composed model formalisms define the rules for structurally joining models.

One of the most important parts of model specification is the definition of what type of information the modeler wants from a model. This process of adding information about what a modeler wants to solve for is called "defining a reward structure." Here we use the term *reward structure* to mean the specification of what information a model solution should provide. A model specified with a reward structure is called a *solvable model*.

Structurally joining models to form composed models requires detailed knowledge of the models' construction. There may be times when such knowledge is not known, or when two models are so too dissimilar that structural composition is not easy; in such cases, the Möbius framework outlines another method of constructing larger models. Models made by this method are called *connected models* and are constructed by connecting models through sharing "solutions." Here we take the term *solution* to mean the value of a reward measure; the value can be a mean, variance, or distribution of a random variable or some combination of these. The modeling formalisms that define the rules that govern the ways models can be constructed in this form are called *connected model formalisms*. Connected models may also yield more efficient model solutions than other model types.

1.4 Research Objectives

The object of our research is to develop a software framework that can serve as the foundation for a modeling tool that facilitates the construction and solution of complex systems. This software framework should be, in part, based upon the Möbius framework as defined in [19] and be realized as a set of C++ base classes. It should also be broad enough to allow for multiple formalism implementations as well as provide the basic constructs for structurally joining

multiple models together. We shall also explore what type of model information is essential in communication between heterogeneous models and between models and solvers.

The framework must not only be able to support many different modeling formalisms, but also be reasonably efficient in both execution time and system resources. In implementing the software framework, we will explore various ways to optimize the base class code.

More specifically, this thesis will define and specify an interface for implementing multiple atomic and composed model formalisms. Such an interface must allow atomic and composed models to communicate information in a generic way to facilitate structural model composition. In particular, the interface should provide a general way of sharing a portion of model state between multiple atomic models. This interface for atomic and composed models must also provide all the necessary information for both simulation-based and analytical/numerical solvers.

In addition, we will look into the issues of extensibility and ease of integration. A successful software framework is not successful unless it is reasonably easy to integrate new formalisms and solution techniques into it. One of the easiest ways to do this is to reduce the number of interactions between models and the software framework. The software framework should also provide as much functionality as possible to make integration easy.

The next chapter will present a brief overview of the Möbius framework, followed by a detailed definition of the Möbius abstract functional interface. The introduction to the Möbius framework is intended to provide background for the abstract functional interface; we encourage the reader to consult [19] for a thorough understanding of Möbius framework concepts. We will first define the abstract functional interface in terms of entities, set-theoretic functions, and entity-specific attributes. These will serve as a basis for our implementation of the abstract functional interface, which is outlined in Chapter 3. We specify the abstract functional interface implementation in terms of C++ classes, methods, and data structures—all of which represent actual code used to implemented the Möbius tool. Chapter 4 specifies how we implement a particular atomic model formalism (stochastic activity networks) within the abstract functional interface specified in the previous chapter. Finally, Chapter 5 reviews our conclusions and cites areas for future work.

CHAPTER 2

THE MÖBIUS ABSTRACT MODEL SPECIFICATION

2.1 Introduction

Möbius is a stochastic, discrete-event, modeling tool that we built around the Möbius modeling framework as defined by Deavours [19]. The Möbius framework is a precise mathematical specification of models, modeling formalisms, reward metrics, and model construction. The main focus of this thesis is on how the Möbius framework can be used to build an extensible modeling tool via an "abstract functional interface." The Möbius tool addresses the issues of allowing multiple formalisms and solution methods within a single tool by requiring all models to communicate information to other models and solvers through the abstract functional interface. The *abstract functional interface* is a set of methods defined on a set of base classes that all models must implement in order to work within the Möbius tool. The abstract functional interface is the mechanism by which we implemented the concept of heterogeneous modeling in the Möbius tool. This thesis defines this abstract functional interface and thus the mechanism by which all models communicate with each other in the Möbius tool. This interface defines the set of operations necessary for implementing any new formalism or solver within the Möbius tool.

In addition to making the tool extensible, the abstract functional interface has the added benefit of providing a means for data encapsulation. In short, this means that formalism implementors are free to implement the abstract functional interface in the most efficient manner using whatever data structures and algorithms they deem appropriate. One important benefit of

the abstract functional interface and data encapsulation is that formalism implementors decide how they want to store and change model state in their formalisms.

The abstract functional interface is not the only way to implement a tool that is capable of building heterogeneous models. For instance, another possible method for implementing a heterogeneous modeling environment would be to translate all models to a universal representation. We decided not to pursue this implementation, because the translation process would undoubtedly remove formalism-specific knowledge of the model. Ideally, we would like to maintain as much formalism-specific knowledge as possible. Such knowledge could later be exploited by specialized solvers. For instance, if we translate a product-form queuing network model to another universal modeling language, we would lose our ability to use specialized queuing network solution techniques like mean-value analysis [20].

The abstract functional interface mainly acts as a communication interface between models and solvers. Solvers built in the Möbius tool communicate with models by calling methods in the abstract functional interface. These methods return generic information about the model and change the model's state. Methods that return generic information about the model can also be used to communicate information between models specified in different modeling formalisms. Therefore, the abstract modeling framework facilitates the construction of heterogeneous models. This feature is of particular interest when modeling large systems, whose scope may encompass many different application domains.

This chapter describes how we built the Möbius modeling tool using the ideas defined in the Möbius framework and implemented them by means of the Möbius abstract functional interface. The Möbius framework outlines some general ideas about how extensible models can be constructed, but does not specify any particular implementation of those ideas. The abstract functional interface design represents a plan for implementing some of the ideas in the Möbius framework in a software framework. Before we describe the abstract functional interface in detail, we must first review the key concepts that define the Möbius framework, because our description of the abstract functional interface uses terminology adapted from the Möbius framework.

2.2 Definition of Symbols

We will use many symbols to define concepts used in the Möbius framework and our implementation of the Möbius tool. The meanings of those symbols are listed in Table 2.1; many of the listed definitions include terms that will be defined in later sections.

Table 2.1 Definition of symbols

2^T	The power set of T (the set of all possible subsets)
S	The set of all state variables in a model
A	The set of all actions
G	The set of all groups
Z	The set of all integers
Z^+	The set of all positive integers
$\Re^>$	The set of all positive real numbers
Ψ	The set of all reachable model states
ψ	A specific model state
Σ	The set of all model states with respect to state variables
σ	A model state with respect to state variables
α	A model state with respect to actions
γ	A model state with respect to groups
$\Sigma_{\nu,\sigma}$	The set of all possible next states from σ
Σ_{a_ϱ}	The subset of model states for which a's reactivation predicate is true
Σ_{a_ϵ}	The set of states in which a is enabled
$A_{\epsilon,\sigma}$	The set of all enabled actions in σ
$g.M$	The set of members for a particular group g
$g.M_{\rho,\sigma}$	The subset of $g.M$ that have the highest rank for state σ
τ	The length of time an action is enabled before being interrupted
θ	An action's sampled time-to-completion
t	A particular state variable type
T	The set of all permissible state variable types in the abstract functional interface

We use boldface type to denote functions that operate on sets of Möbius entities. Typewriter type is used to denote implementations of functions and methods in a programming language.

If an action or set of actions has the subscript ϵ, it means that all the actions in question are enabled. If an action or a group has the subscript ρ, it means that the action has the highest-rank value in the current model state.

2.3 Overview of the Möbius Framework

The Möbius framework defines all models in terms of basic entities. These *entities* are "state variables," "actions," and "groups." These three entities (which we define later) are the building blocks for all models, including atomic, composed, and connected models. Each entity contains a portion of the model state and defines a set of functions.

A *state variable* is the Möbius entity used for storing system state. State variables have two functions defined on them: **type** and **value**. The **type** function maps the state variable to a specific set of possible "values." The set of all possible types is defined in Table 2.2, in which Z refers to the set of integers, $\Re$ refers to the set of all real numbers, and S is a reference to a state variable. The **value** function corresponds to the "state of the state variable." The value function returns an element from the state variable's value domain, which is defined by the state variable's type. State variables are the primary means of storing the state of a system. The state of all the state variables in the model is denoted as σ.

Table 2.2 Rules for constructing all state variable types in the Möbius framework

$Z \in T$.
$\Re \in T$
$S \in T$.
If $t \in T$, then $2^t \subset T$.
If $t \in T$, then $2^t \in T$.
If $t_1, t_2 \ldots, t_n \in T$, then $t_1 \times t_2 \times \ldots \times t_n \in T$.

Actions are the fundamental Möbius entities used to change the values of state variables and thus the state of a model. Actions are the only entities in the Möbius framework that can change the values of state variables. Petri net transitions, SAN activities, and queuing network servers are all different realizations, in specific formalisms, of the abstract action entity. Since actions must be generic to all modeling formalisms, there are no restrictions on how an action can change the state of a model's state variables. The formalism and the specific model definition define the way an action changes state.

Each action is uniquely defined by its set of action functions (see Table 2.3). Some of these functions are "predicates." In this context, a *predicate* is a Boolean function expressed in terms of the state of a model's state variables. These action functions provide all the information necessary to specify an action's enabling conditions, its state-dependent "firing" time distribution, and the state change function itself. Here we use the term *fire* to mean a specific change of a model's state variables defined by the action.

The exact meanings of the action functions are defined in [19], but we will briefly introduce each action function here to help explain its purpose. The **Fire** function defines how a model changes state when an action fires. The **Fire** function changes the value of the state variable state. The term Σ is the set of all possible model states with respect to state variable values. Each element in Σ is an ordered set of state variable values that corresponds to a possible state of a model's state variables. The **Enabled** function defines the states in which the action can fire. **WorkPolicy** defines how the action behaves if it becomes enabled but does not fire. If an action can fire, then the **Delay** function defines a firing time distribution. In some cases, an action that is enabled and does not fire is viewed as doing "work"; the manner in which the action is affected by previous work is defined by the **Work** function. An action's **Rank** function is used to define a priority-based execution policy, and its **Weight** is used to define a probabilistic execution policy.

The Möbius framework incorporates the interesting and useful concept of "action interruptions." *Interruptions* are changes to an action's state or firing time distribution other than the action becoming disabled. More precisely, an action's state (α) contains a value (stored as a natural number) called the action's "interruption state." The value of this interruption state and the action function **InterFunc** determine how an action is interrupted. The value of the interrupt state is determined by another action function, **InterPred**. **InterPred** is evaluated when the action becomes enabled or interrupted. The **InterFunc** function results in one of four interrupt actions: None (do not change the behavior of the action), Reset (action behaves as if it has just become enabled), Age (the action saves the amount of "work" done), or ChangeIntr (changes the interrupt state to some other value).

Table 2.3 Action functions defined by the Möbius framework

Function Name	*Function Definition*
WorkPolicy	$\Sigma \rightarrow \{\text{Enable, Age}\}$
Enabled	$\Sigma \rightarrow \{\text{true, false}\}$
Delay	$\Sigma \rightarrow (\Re^{>} \rightarrow [0,1])$
Work	$\Sigma \rightarrow (\Re^{>} \rightarrow [0,1])$
InterPred	$\Sigma \rightarrow Z$
InterFunc	$\Sigma \times Z \rightarrow \{\text{None, Reset, Age, ChangeInt}\}$
Rank	$\Sigma \rightarrow Z^{+}$
Weight	$\Sigma \rightarrow \Re^{>}$
Fire	$\Sigma \rightarrow \Sigma$

"Groups" are another Möbius framework entity. A *group* is a set of actions or groups, referred to as the group's *members*, that have a specialized execution policy. The group can be defined so as to allow its members to compete or cooperate in a specialized way. More specifically, a group can reduce the number of states in which a member action can fire, or reduce the probability of a member firing. A group controls the execution policy of its members by implementing a "selection process." A *selection process* is an algorithm for determining which group members can fire in a given state. Groups also maintain state, much as actions do. The group state (denoted as γ) contains information about what members did in past states. More specifically, group state stores information about which members were selected in the previous states. The Möbius framework defines a set of functions on groups. These functions are outlined in Table 2.4.

We will briefly introduce each of these functions (see [19] for more details). The **Members** set associated with each action defines the set of actions and groups that constitute the group's members. The **Actions*** set associated with each action defines the set of group members that are actions. In a manner similar to the action function **Enabled**, the group function **Enabled** defines whether there is an enabled member in the current state. The **Select** and **Select*** functions define how individual members are selected from the set of enabled members. Finally, **ReselPred** and **ReselFunc** define the process by which a group reselects another group member.

Table 2.4 The set of functions and sets defined on groups in the Möbius framework

Name	*Definition*
Members	$2^{A \cup G}$
Actions*	2^{A}
Enabled*	$\Sigma \rightarrow \{true, false\}$
Select	$\Sigma \times$ Members $\rightarrow \Re^{>}$
Select*	$\Sigma \times$ Actions$* \rightarrow [0, 1]$
ReselPred	$\Sigma \rightarrow Z$
ReselFunc	$\Sigma \times Z \rightarrow \{None, ReselReset, ReselAge, ChangeResel\}$

Given these three fundamental entities, the Möbius framework defines a model. A *model* is a collection of state variables, actions, groups, all the functions defined on them, and the names of all the entities. A *model state* contains state variable state, action state, and group state to form the tuple ψ = (σ, α, γ). Here σ is the value of all the state variables, α is action state of all the actions, and γ is the group state of all the groups for a model state ψ.

2.4 Möbius Abstract Functional Interface

Now that we have summarized the important parts of the Möbius framework, we will explain how these concepts were realized via an abstract functional interface. The following subsections outline how each of the basic framework entities have been incorporated into the abstract functional interface. In many cases, for ease of implementation we created an abstract functional interface different from the Möbius framework.

The following sections outline the design of the Möbius abstract functional interface and the motivations behind the design details. Specific details of how this abstract functional interface was realized in software is outlined in the next chapter.

2.4.1 State variables in the Möbius abstract functional interface

Central to both the Möbius framework and the Möbius tool is the concept of a "state variable." A *state variable* is the basic state-storing entity in Möbius. As in the Möbius framework,

we define a state variable by its type and its value. However, in the abstract functional interface, we define a state variable's type to be a fixed attribute of the state variable. We will refer to the type of a state variable s using the syntax $s.t$, where t is one of the allowable types T (where T is the set of all allowable types). Thus a state variable's type defines the set of values that the state variable can hold. We say a state variable of type $t \in T$ can hold a value v if v is a member of t. The set of all possible types in the abstract functional interface (T) can be constructed using the following rules:

(1) bool, char, int, float, double, short $\in$ T (basic type rule; see Table 2.5)

(2) if $t \in T$, $2^t \subset$ T (subset rule)

(3) if t_1, t_2, t_3,..., $t_n \in T$, then $t_1 \times t_2 \times t_3 \times t_n \in$ T (structured type rule)

Table 2.5 Definition of basic types in the abstract functional interface

State Variable Type	*Permissible Values*
bool	0 (false) or 1 (true)
char	-128 to 127
int	-2147483648 to 2147483647
float	32 bit number
double	64 bit number
short	-32768 to 32767

These rules allow one to create complex representations of a model's state by creating structured state variables. The existence of structured state variables allows for formalisms with rich notions of state.

As in the Möbius framework, a state variable may have one or more *value functions*. In general, a value function returns the value of a state variable in a particular model state, i.e.,

$$value : S \rightarrow t \text{ such that } t \in T.$$

The nature and the implementation of these value functions are specific to the choice of state type within a formalism. In the Möbius tool, we further distinguish between the con-

cepts of primary and secondary value functions of a state variable. The *primary value function* ($value_p(s)$) defined on a state variable is the value function that is used to define "state equivalence" for a state variable. A *secondary value function* ($value_s(s)$) is a value function defined on a state variable used to simplify model specification and to create "functional sharing" among two or more state variables (defined in Section 3.1.2).

A state variable's primary value function is a value function whose range is defined by the state variable's type, i.e.,

$$value_p(s) : S \rightarrow t \text{ such that } t \in T \text{ and } t = s.t \; \forall s \in S.$$

Because the primary value function defines state variable state equivalence, we can say that σ and σ' are in *equivalent states* with respect to s if $value_p(s) \mid_\sigma = value_p(s) \mid_{\sigma'}$. Here, $value_p(s) \mid_\sigma$ denotes the primary value of s in state σ. We will use this notion of state variable equivalence to define other concepts that are central to the abstract functional interface.

A secondary value function cannot be used to define an equivalence relationship. A secondary value function does *not* have to return a value in the domain of $s.t$, but the type of value returned by a secondary value function must be an element in T.

$$value_s(s) : S \rightarrow t \text{ such that } t \in T, t \text{ is not necessarily equal to } s.t.$$

One might define a secondary value function on a state variable to make reward specification easier or as a means of creating a functionally shared state variable (see Section 3.1.2). All of the secondary value functions must satisfy the following condition:

$$value_p(s) \mid_\sigma \; = value_p(s) \mid_{\sigma'} \Rightarrow value_s(s) \mid_\sigma \; = value_s(s) \mid_{\sigma'} .$$

This condition simply states that if a state variable s has the same primary value in states σ and σ', then all secondary values must be the same in states σ and σ'. The reasoning behind this condition is that the primary value function determines whether a state variable's value is equivalent in two different model states. If a state variable's value is equivalent in two different

model states, we can say that the state variable is "unchanged" by a state transition from σ to σ'. If any of the state variable value functions were different in states σ and σ', that would indicate that something has changed the state variable. Thus, imposing the previously defined condition allows us to evaluate whether a state variable is "unchanged" in two states without having to evaluate all the state variable functions in both states. Using the previous equation, we know that if the primary values are the same in the two states, then all the state variable value functions are the same in the two states.

The concept of primary and secondary value functions may be illustrated with an example. Suppose we create a state variable to represent the state of a computer as a subset of integers (using rules 1 and 2 defined in this section) such that the values shown in Table 2.6 correspond to actual computer states.

Table 2.6 An example state variable

State Value	*Computer State*
0	idle
1	working
2	dead

The primary value function would be the identity function (this is almost always true for primary value function implementations). An example of a secondary value function would be a function **alive** that returns 0 if the state value is 2, and otherwise returns 1.

Introduction to state in the Möbius tool. State variables constitute the main building blocks of a model's state in the abstract functional interface. The Möbius framework describes the state of a model in terms of the tuple (σ, α, γ), which includes action state (α) and group state (γ), in addition to state variable state (σ). The abstract functional interface does includes a slightly different notion of action and group state than the Möbius framework. The implementation of the abstract functional interface is based upon assumptions on how this state will be used by the solvers. The differences between the Möbius framework and the abstract functional interface on the issues of action state and group state are explained in their respective sections.

However, the net result of these assumptions is that the abstract functional interface defines a model's state solely by the state variable state σ. Consequently, we use the notion Σ to denote all of the possible states of a model with respect to state variable.

2.4.1.1 Sharable state variables

Earlier we defined a set of models called "composed models." These are models that one constructs by structurally joining two or more models together. One of the simplest ways to structurally join two models together is to share a common portion of state. For this reason, the abstract functional interface supports state-sharing among models. We can use the concepts of primary and secondary value functions to define how models are constructed through shared state.

Sharing state in the abstract functional interface requires that a state variable (or a portion of a state variable) in a model be joined to another state variable (or a portion of another state variable) through a "sharing relationship." A *sharing relationship* is a redefinition of a state variable's primary value function in terms of another state variable's value function (not necessarily a primary value function). We say that the two state variables joined together are members of the same "sharing set." A *sharing set* is a set of state variables that share a common state value.

There are several types of sharing relationships supported by the abstract functional interface. If the primary value function of a state variable (or a portion of a state variable) is replaced with another state variable primary value function, then the two state variables are said to be connected through equivalence sharing. Therefore, an *equivalence sharing* relationship is one in which a state variable's primary value function is replaced by another state variable's primary value function. A state variable whose primary value function is replaced with a secondary value function creates a *functionally sharing* relationship.

2.4.2 Actions in the Möbius abstract functional interface

The way we define actions in the abstract functional interface is very closely tied to the action definition provided by the Möbius framework. In the following sections, we define the meanings of all the action functions for the abstract functional interface. These action function definitions play an important part in defining the execution policy for the abstract functional interface. Note that they are somewhat different from those used in the Möbius framework; thus, the execution policy is also different from the Möbius framework. Table 2.7 summarizes the action functions for each action in the abstract functional interface. Each of these functions will be described in a following subsection. In addition to functions, each action has a set of attributes that further define its operation. These attributes are listed in Table 2.8.

Table 2.7 Functions defined on actions

Function Name	*Function Definition*
Enabled	**Enabled** : $\Sigma \rightarrow \{\text{true, false}\}$.
Fire	**Fire** : $\Sigma \rightarrow \Sigma$.
ReactivationPredicate	**ReactivationPredicate** : $\Sigma \rightarrow \{true, false\}$.
ReactivationFunction	**ReactivationFunction** : $\Sigma \rightarrow \{true, false\}$.
SampleDistribution	**SampleDistribution** : $\Sigma \rightarrow (\Re^{>} \rightarrow [0, 1])$.
Rank	**Rank** : $\Sigma \rightarrow Z^{+}$.
Weight	**Weight** : $\Sigma \rightarrow \Re^{>}$.

Table 2.8 Action attributes

Action Attribute	*Description*
Name	The name of the action
DistributionType	The type of probability distribution used to define the firing time delay
ExecutionPolicyType	The the type of action execution policy
GroupID	The highest-level group to which the action belongs
EnablingStateVariables	The list of state variables whose state variable state defines the action's **Enabled** function
AffectedStateVariables	The list of state variables whose state variable state is affected by the action's **Fire** function

2.4.2.1 The Enabled function

The **Enabled** function determines in which states an action can fire. The **Enabled** function is a Boolean expression that evaluates each model state as either true or false for purposes of firing:

$$\textbf{Enabled} : \Sigma \rightarrow \{\text{true, false}\}$$

An action's **Enabled** function plays a very important role in specifying when a model changes state. An action's *enabling states* are a set of model states in which the action can change the state of the model. The term Σ_{a_ϵ} denotes the set of enabling states for an action a. We can more formally define the set of enabling states of a in terms of a's **Enabled** function:

$$\Sigma_{a_\epsilon} = \{\sigma \in \Sigma : a.\textbf{Enabled}(\sigma) = \textit{true}\}.$$

The set of enabling states is the subset of all possible state variable states Σ in which a's enabling predicate is true. The **Enabled** function is also used to define another concept used throughout the abstract functional interface: the *set of enabled actions*, which is the set of actions whose **Enabled** functions are true in a given state. We can formally define the set of enabled actions for state σ (denoted as $A_{\epsilon,\sigma}$) as

$$A_{\epsilon,\sigma} = \{a \in A : a.\textbf{Enabled}(\sigma) \rightarrow true\}.$$

An action's **Enabled** function allows us to specify the conditions under which a specific state change can occur. The result of the firing of all the enabled in highest-ranked group actions is a set of possible next states for σ, denoted by $\Sigma_{\nu,\sigma}$.

Because an action's **Enabled** function is a function of the state variable state σ, we can define the subset of state variables that are used in the definition of an action's **Enabled** function:

$$S_{a_\epsilon} = \{s \in S : \exists\, \sigma \notin \Sigma_{a_\epsilon}, \exists\, \sigma' \in (\Sigma_{a_\epsilon} \cap \Sigma_{\nu,\sigma}) \text{ such that } value_p(s) \mid_\sigma \neq value_p(s) \mid_{\sigma'}\}.$$

S_{a_ϵ}, as defined here, is actually a superset of the enabling state variables because there may be state variables whose values change but whose values are not part the state variable's **Enabled** function definition. Note that $\Sigma_{a_\epsilon} \cap \Sigma_{\nu,\sigma}$ is the set of model states that are reachable from σ and in which a's enabling predicate is true. This set of state variables, S_{a_ϵ}, defines the action attribute EnablingStateVariables.

2.4.2.2 Fire

Given that an action is enabled, it may fire according to the action's execution policy.

$$\textbf{Fire} : \Sigma \rightarrow \Sigma.$$

An action's firing may change the value of certain model state variables. The set of all state variables affected by an action's firing is called the action's *affected state variables*. The set of affected state variables for an action a is defined as follows:

$$S_{a_\phi} = \{s \in S : \exists\, \sigma \in \Sigma_{a_\epsilon}, \sigma' \in \Sigma_{\nu,\sigma}, \text{ such that } value_p(s) \mid_\sigma \neq value_p(s) \mid_{\sigma'}\}.$$

An ordered sequence of individual action firings will result in a sequence of model state changes. We call this sequence of state changes a *trajectory* through the model's state space. The set of all such sequence represents the set of all trajectories through the model's state space.

2.4.2.3 ReactivationPredicate

The abstract functional interface implements the concept of action interrupts differently from the Möbius framework. The abstract functional interface prescribes that an action's reactivation state is saved as a Boolean value. (The reactivation state value is part of the action's state α). The Boolean value indicates whether an action is "interruptible." If an action's interrupt state is true, then the action can be interrupted at future state changes if it is still enabled.

In contrast, the Möbius framework saves the action interrupt state as an integer, and thus has the ability to encompass more complicated interrupt policies. The decision to implement a reduced version of the action interrupt state was motivated by its ease of implementation.

The value of the abstract functional interface action interrupt state is determined by the action's **ReactivationPredicate** function:

$$\textbf{ReactivationPredicate} : \Sigma \rightarrow \{true, false\}.$$

The **ReactivationPredicate** function is evaluated in every state in which the action's **Enabled** function is true, if the action's **Enabled** function was false in the previous state. Additionally, an action's **ReactivationPredicate** is evaluated when the action fires and remains enabled in the new state. These two conditions define the set of states in which the action is "activated."

The Möbius framework defines an action function, called the **InterruptPredicate**, which is similar to the abstract functional interface's **ReactivationPredicate**. However, there is an important difference between the two action functions. The **InterruptPredicate** is a mapping from a model state to the set of integers. The **ReactivationPredicate** is a mapping from model state to $\{true, false\}$. A "true" value indicates the action might be restarted depending on its **ReactivationFunction** (see below). In the abstract functional interface, an action whose **ReactivationPredicate** is true is "restartable." The implementation in the abstract functional interface thus requires that the action function **ReactivationFunction** (similar to the Möbius framework's **InterruptFunction**) be evaluated only if the action is restartable. The Möbius framework specification for the interruption process is richer than the abstract model specification, because it allows multiple interruption states; nevertheless, implementing a subset of the functionality in the abstract functional interface leads to better efficiency.

2.4.2.4 ReactivationFunction

The abstract functional interface action function **ReactivationFunction** was motivated by the Möbius framework action function **InterruptFunction**. Both action functions have the

same purpose: to determine when and how an action should be interrupted. In the abstract functional interface, if the action's interrupt state is true, then the **ReactivationFunction** is evaluated at every subsequent state change. **ReactivationFunction** yields a Boolean value that is used to determine if the action should be interrupted:

$$\textbf{ReactivationFunction} : \Sigma \rightarrow \{true, false\}.$$

The action taken upon interrupt (see Table 2.9) depends upon the action's execution policy type (an attribute of actions described in Section 2.5).

Table 2.9 Possible outcomes of evaluating an action's **ReactivationFunction** given that **ReactivationPredicate** was true at activation time

(ExecutionPolicy, ReactivationFunction)	*Outcome*
(Resample, True)	Reset
(Resample, False)	Reset
(Enabled, True)	Reset
(Enabled, False)	None
(Age, True)	Age
(Age, False)	None

2.4.2.5 Rank

Rank is a function defined on actions that allows the modeler to implement priority-based ordering of actions scheduled to fire at the same time. An action's rank value can also be used to define a priority-based preselection algorithm for action groups (see Section 2.4.3). A priority-based selection policy implemented over a set of actions can replace a race-based execution policy.

Each action has a state variable state-specific integer rank value that represents its priority level for that enabling state. Higher numerical values imply higher priority, with 1 being the lowest priority an action can have. Thus, the definition of action **Rank** is analogous to the definition given in the Möbius framework, i.e.,

$$\textbf{Rank} : \Sigma \rightarrow Z^{+}.$$

2.4.2.6 Weight

In order to resolve selection among similarly ranked actions, a probabilistic algorithm can be used to select a specific action. Each action has a weight function that determines the action's weight for any given enabling state:

$$\textbf{Weight} : \Sigma \rightarrow \Re^{>}.$$

Greater weight values imply that an action is more likely to fire. When used in action groups (defined in Section 2.4.3) an action's weight is used to calculate the probability of selecting a representative member from a set of simultaneously enabled, equally ranked members.

2.4.2.7 SampleDistribution

SampleDistribution is an action function defined by the abstract functional interface that is used to facilitate model simulation. In any given state for which the action is enabled, the action's time to completion is a random variable. When using simulation to solve a model, we randomly sample from the action's time-to-completion distribution. One may think of an action's time-to-completion in two different ways. It may be viewed as the time it takes an action to complete a task it starts working on when it becomes enabled. Alternatively, time-to-completion may be viewed as a scheduled event that does not correspond to a task the action is working on. It is merely a scheduled state change. We will refer to the first view of time-to-completion as the "work-centric" notion, and the second view as the "event-centric" notion.

An event-centric notion of time-to-completion corresponds to actions that represent parts of a system that do not have a clear notion of work. Such an action, when enabled, schedules a state-changing event for some time in the future. These events may not correspond to any notion of an "underlying process" that has a concept of partial completion. The event may simply represent an explicit state change scheduled at a specific time given that it is not disabled

before that specified time. Such an action would not need to remember how close it came to its scheduled execution time. An example of an event-centric time-to-completion is an acknowledgment timer that schedules a retransmission of data for a specific time in the future unless the timer receives an acknowledgment.

The work-centric view of the time-to-completion sees an action as performing work on a task starting from the time it becomes enabled, and ending when the task is completed. Completion of the task signals that the model is ready to change state. Given that some actions represent underlying concepts of tasks, it may be beneficial to have some way of storing the amount of work done by an action in the event that the action becomes disabled before completion. The amount of work done could be saved as a value of a state variable, if that state variable could change as a result of an action becoming disabled. However, a state variable's state can change only because of an action's firing. This would require the firing action to save the amount of work done by all simultaneous enabled actions that become disabled in the new state. This is not a convenient way of implementing a work-centric actions, therefore the amount of work done is not (by default) saved as part of the state variable state (σ), it is saved as part of the action state (α).

The next important question is how we wish to quantify the amount of work done by an action before it becomes disabled. One way to measure the amount of work done is to look at the fraction of time the action was enabled compared to the sampled time-to-completion.

Given that **FractionComplete** fraction of the work has been done before the action becomes enabled, the action is enabled for time τ, and the sampled time returned by the action's sample distribution is θ, the fraction of work done when the action becomes disabled is

$$\textbf{FractionComplete}' = \textbf{FractionComplete} + (1 - \textbf{FractionComplete})\frac{\tau}{\theta}$$

Assuming that the completion time distribution of an action is not a function of the fraction of work completed, the amount of work done during a previous enabling period can be used to reduce the completion time the next time the action is enabled. More specifically, the action state **FractionComplete** is used as a linear scaling factor for the abstract functional interface

method **SampleDistribution**. To do this, **SampleDistribution** method scales the sampled value of the random variable (θ) by a linear scaling factor (**FractionComplete**). The scaled value returned by **SampleDistribution** is calculated by the formula

$$\theta' = (1 - \textbf{FractionComplete})\theta$$

. **SampleDistribution** yields a random variable that corresponds to the completion time of an action, which is a function of the model's state when the action becomes enabled.

$$\textbf{SampleDistribution} : \Sigma \rightarrow (\Re^{>} \rightarrow [0, 1])$$

Here θ represents a sample from the action's time to completion distribution. The value θ is scaled according to the amount of work still to be done $(1-$**FractionComplete**$)$. This results in a value θ' that is no greater than the original value θ.

The Möbius framework has a more sophisticated scheme in which the actions save the amount of "work" done during a previous enabling state. In it, the amount of work is determined by saving the value of the distribution function at the disabling time. In the framework, the work done by an action by some time τ (the time at which an action becomes disabled) is the probability that an action would have fired by time τ. This value is equal to the value of the probability distribution function at τ. (For the sake of brevity, we have omitted details about how nonstrictly increasing distribution functions are dealt with; see [19] for a complete explanation.) This work value is used to determine a new firing time distribution based upon the amount of work done in a previous enabling state. Note that this is different from using a simple linear scaling factor.

2.4.2.8 Action state

The abstract functional interface represents the state of an action with three different quantities. The enabling status (true or false) of an action in the previous state constitutes the action state member **Disabled**. This portion of action state is used by a group (see Section 2.4.3) when

it chooses its representative member. Section 3.2 discusses how this portion of action state is used by certain types of groups.

The second portion of action state (**Reactivation**) saves the value of the reactivation predicate. This part of action state must be saved so that future states can correctly determine when an action should be restarted.

The third portion of action state is **FractionComplete**. **FractionComplete** is a fraction and thus should always have a value in the range [0,1). **FractionComplete** represents the amount of work the action completed in all previous enabling states. As just described, this value is used by solvers to scale the amount of time the action is enabled before it fires in future enabling states.

2.4.3 Groups in the Möbius abstract functional interface

In addition to state variables and actions, the abstract functional interface also defines another fundamental construct: the "action group." We will use the term group to mean "action group." A *group* is a set of actions and/or groups that cooperate or compete amongst themselves. The nature of a group's cooperation or competition is encapsulated in the group's "selection algorithm." A group's *selection algorithm* refers to the process that a group uses to define which one of its members is the group's *representative* in a given state. Only group "representative" members are able to fire in a given state. Thus, groups allow a formalism to implement a non-race-based execution policy (see Section 2.5) among a subset of actions based upon a group selection algorithm. This selection algorithm can be either priority- or probability-based. The actions (or groups) that belong to a group are called the group's *members*.

Groups can be categorized into two main subdivisions, which differ in the implementation of their respective selection algorithms: preselection groups and postselection groups. *Preselection groups* are groups that choose their representative actions at activation time, whereas *postselection groups* choose their representatives at firing time.

Groups implement a superset of the functions defined on actions. The two additional functions that groups implement are **Select** and **Probability**. The **Select** function is the algorithm used by the group to select a unique member to be the group's representative action. Group members include actions and other groups (see Section 2.4.3.3). If the selected member is a group, then the **Select** function is called on the selected group until the selected member is an action.

$$\textbf{Select} : \Sigma \rightarrow A \cup G.$$

Groups must also implement a method to determine the probability that an enabled member of the group will fire in a given state. The **Probability** function determines this value:

$$\textbf{Probability} : \Sigma \times A \cup G \rightarrow [0,1].$$

Because this probability is a function of a specific selection algorithm, it must be defined for each unique selection policy. The abstract functional interface provides a default selection policy that uses both rank and weight values of the enabled group members:

$$g.M_{\epsilon,\sigma} = \{m \in g.M : m.\textbf{Enabled}(\sigma) \rightarrow true\}.$$

Here m is a member of group g that contains a set of members $g.M$. Within this set of enabled group members ($g.M_{\epsilon,\sigma}$), the set $g.M_{\rho,\sigma}$ contains the members that have the highest rank value for state σ. The set $g.M_{\rho,\sigma}$ is defined as follows:

$$g.M_{\rho,\sigma} = \{m \in g.M_{\epsilon,\sigma} : m.\textbf{Rank}(\sigma) = \max(\{m.\textbf{Rank} : m \in g.M_{\epsilon,\sigma}\}).$$

Using this notation one can define how the abstract functional interface calculates the probability of selecting member m in σ:

$$\text{Prob}(m,\sigma) = \frac{m.\textbf{Weight}(\sigma)}{(\sum_{m \in g.M_{\rho,\sigma}} m.\textbf{Weight}(\sigma))}. \tag{2.1}$$

The default implementation uses a priority-based scheme that gives probability mass only to members in the highest-ranked group. The member weight values are used to determine the amount of probability mass to assign to each group member. Based upon this algorithm, the group can construct a distribution function. This distribution function can then be used by the **Select** method to probabilistically select the group's representative action.

2.4.3.1 Preselection groups

Preselection groups represent a class of groups in which each group selects its representative action based upon "reselection conditions." *Reselection conditions* are rules that determine when a group selects its representative member. Because groups can choose their selected members at different times, the reselection conditions are determined by the type of group. The Möbius framework further divides preselection groups into two subcategories based upon how the reselection conditions are defined. These two subgroups are "variable preselection groups" and "persistent preselection groups." *Variable preselection groups* are preselection groups whose reselection conditions state that the group should select a member if there has been a change in the enabling conditions of one or more members, and if at least one member is enabled in the new state. A change in a group's enabling conditions implies that there is a change in the enabling status of at least one group member. The reselection conditions for a *persistent preselection group* require that the group reselect a representative member in states in which no group member was enabled in the previous state, and at least one member is enabled in the current state. Persistent preselect groups also reselect when the representative member fires and the group is still enabled in the new state.

The abstract functional interface, unlike the Möbius framework, does not support variable preselection groups. Implementing variable preselection groups requires a more complex tool design. Because preselection groups are not a frequently used aspect, it was deemed appropriate to implement only persistent preselection groups.

2.4.3.2 Postselection groups

Postselection groups are the other main subdivision of action groups. The representative action in a postselection group is selected at firing time. In order for this selection policy to make sense, every member of a postselection group must have the same firing time distribution, or a joint distribution must be specified for the group for every possible subset of simultaneously enabled members.

The abstract functional interface makes the assumption that all actions in a postselection group have the same firing time distribution, and that the group is thereby free to use any member's distribution function to determine the group's scheduled completion time. The abstract functional interface also places further restrictions by requiring that all actions in a postselection group have the same enabling conditions. This prevents the model from ever changing state in such a way that the member's enabling conditions change but the group remains enabled. The abstract functional interface does not specify how the representative member selection should be done in the case where the enabling set of members has changed during the enabling lifetime of the group. To prevent changes in the enabling membership while a postselection group is enabled, the abstract functional interface also requires that all the members' reactivation predicates and functions also be identical. This prevents two enabled members from the same group from having different firing time distributions over the same interval. Resampling one of the member actions during enabling time would affect that action's distribution unless all the group members have exponential distributions, in which case resampling does not affect the firing time distribution.

We can summarize the restriction on postselection group members by saying that all action functions of all members in a postselection group must be identical, except for **Rank**, **Weight**, and **Fire**.

2.4.3.3 Multilevel groups

Groups are an intrinsic part of the abstract functional interface, because all actions are required to be members of at least one group. Groups can contain members that are other groups as long as the member group contains a subset of actions contained in the parent group.

In general, preselection groups can be nested within each other until all of a group's members are actions. This is possible because the preselection process can be performed recursively through many layers of groups. Those groups with members that are preselection groups would first select a representative group (in the same way an action is selected), and then the selected group would select a member from its group.

A postselection group can be a member of a preselection group, but a preselection group cannot be a member of a postselection group, because preselection groups do not have the same restrictions on action functions that postselection groups do. A preselection group is likely to have members with different enabling predicates and time-to-completion distributions. A postselection group could theoretically be a member of another postselection group, but all the members of the contained postselection group would need to have the same action methods as the other members. Thus, having multiple levels of postselection groups is unnecessary, since the same functionality can be achieved in a single postselection group. For this reason the abstract functional interface does not support multiple levels of postselection groups.

2.4.4 Models in the Möbius abstract functional interface

Now that we have defined state variables, actions, and groups, it is possible to define models in the abstract functional interface. A model describes the behavior of a system. The behavior of the system is specified in terms of the system's state and how the system changes as a function of system state and time. At the most basic level, a model is essentially a container for state variables, actions, and groups. Like state variables, each model has a type. A model's type is the specific modeling formalism in which the model is implemented.

The state of the simplest types of models is represented by the collective state of the model's state variables, actions, and groups. More complex models can contain other models when

two or more models are composed together through a process of model composition [21]. Nevertheless, all models can be recursively decomposed to obtain a final set of state variables, actions, and groups that completely represent the "highest-level model." The *highest-level model* is a model that is not contained in any other model. The state of a model is constructed from the sets of state variables, actions, and groups contained within the highest-level model. The state of all three sets forms the tuple (σ,α,γ), which is the model's state. The way a system changes state as a function of time and state is described in the functionality of a model's actions and groups.

In addition to being a container for state variables, actions, and groups, models must also be able to perform certain operations on their contained Möbius entities. These operations are realized as a set of functions called the *model functions*. These model functions provide the same types of functionality as action functions (see Table 2.7 on page 18) and groups functions (see Section 2.4.3). These model functions represent a key design aspect that makes heterogeneous models possible. They communicate important information about the model to other models and solvers. There are two main categories of model functions: those that provide generic information about the model's structure and those that operate on the model (see Table 2.10).

Table 2.10 A list of model functions

Function Name	*Function Definition*
ListActions	**ListActions** $\rightarrow A$
ListGroups	**ListGroups** $\rightarrow G$
ListStateVariables	**ListStateVariables** $\rightarrow S$
SetState	**SetState**: $\Sigma \rightarrow \Sigma$
CurrentState	**CurrentState** $\rightarrow \Sigma$

ListActions, **ListGroups**, and **ListStateVariables** all return a set of Möbius entities that correspond to their respective function names. Solvers use both **ListActions** and **ListGroups** to look at all the actions and groups in a model. The **CurrentState** and **SetState** functions are used to read and write the model state. These functions are important for resetting a model's

state in state-space exploration. There are also used to reset the model state between batches in simulation-based solvers.

The functions that return information about model structure do so by returning Möbius entities (see Section 2.3) in the models. Because every model must be implemented using these Möbius entities, they represent a natural way to exchange information among models. However, this does not mean that models expressed in different modeling formalisms can determine everything about each other. For instance, these model functions do not provide any explicit information about how a formalism changes the model state or what types of formalism-specific methods are defined on state variables.

2.5 Execution Policy

One important part of building stochastic models is the precise definition of the rules that govern how state changes as a function of time. We will refer to this set of rules as the model's *execution policy*. A model's execution policy is particularly important in stochastic modeling, because event completion times are usually described in terms of continuous distribution functions. This means that each of the enabled actions can fire at one of many (possibly infinite) different time-points. In practice, it is often convenient to add more complexity to the execution policy to allow for greater flexibility of state changes. The Möbius model execution algorithm is specific to the solver and examples of such algorithms are outside the scope of this thesis. However, all of these solver-specific algorithm rely upon the methods in the abstract functional interface. A simulation-based algorithm is given in [22] and a state space generation-based approach is given in [23].

Much of the model execution policy centers around how individual actions view their time-to-completion distribution (see Section 2.4.2.7). We use the term "work policies" to refer to the different ways in which actions view their time to completion. Since actions in a model can have different work policies and an action's work policy defines how the action will be have in the future, we can say that each action has an "action execution policy." An *action execution policy* defines the behavior of an action due to a specific work policy.

Before we describe the Möbius action execution policy, we must first define some terms. Two actions are said to be *racing* if they are both enabled in the same state and either one can fire first. The next state change is determined by the action with the minimum completion time.

Priority is a rule sometimes used in stochastic modeling to explicitly define an ordering of events or "event generation." *Event generation* means the rules used to define which of the enabled actions are able to generate events in a particular state. We will use this concept to define a subset of enabled actions in a state called the "event-generating actions."

Möbius supports three race-based action execution policies for actions. Nevertheless, this does not completely describe the model execution policy, because each group prescribes a priority or probabilistic "selection policy" among its members. This selection process affects the way a model can change state. A *selection policy* is a rule used to define the model's execution policy. A selection policy defines a subset of enabled actions in a group called *event-generating actions*, which are actions whose **Enabled** functions are true, and that are selected by the group to which they belong. An *event* is a scheduled state change. Enabled actions that are not event-generating actions cannot change a model's state.

Next we will look at the three race-based action execution policies supported in the abstract functional interface. Every action has an execution policy that defines how the action behaves when the model state changes before the action fires. The simplest action execution policy involves restarting the action in every model state for which it is enabled. In this execution policy, all other enabled actions lose the work done up until that point. This policy is referred to as "Race-Resampling." Another supported execution policy is "Race-Enabled," in which an action is not restarted unless it becomes disabled in the new state. In this policy, work done in the past is preserved until the action fires or it becomes disabled. The last action-execution policy supported is "Race-Age," in which the action saves the amount of work done in a previous enabling state as part of action state. The amount of work done previously is factored into the calculation of the action's expected completion time when, and if, it becomes enabled in the future. We will now describe the operation of each of these execution policies in more detail. These execution policies were motivated by work done by Marsan et al. [24].

However, the execution policies described by Marsan are applied to an entire model, whereas race-based policies are assigned in an action-by-action basis for Möbius models.

2.5.0.1 Race-Resample

Race-Resampling is the simplest of all race-based execution policies. Given a set of simultaneously enabled actions, a Race-Resample action will do one of two things at the next scheduled state change. If the action's event has caused the state change, then the action will execute its **Fire** method as usual. If the Race-Resample action has not completed yet, then the action will lose all the work it has done up until that point. If the action is still enabled in the new state, its time-to-completion distribution is a random variable that is a function only of the new state of the model.

2.5.0.2 Race-Enabled

Race-Enabled is a similarly simplistic execution policy for actions. If a Race-Enabled action is still enabled in the new state, the action will continue to perform work in accordance with the action's time-to-completion distribution at enabling time. If the action is not enabled in the new model state, a Race-Enabled action does not save the amount of work done up until that point. The next time the action is re-enabled, it acts as though no previous work has been done; thus, its time-to-completion distribution is only a function of the state variable state. This execution policy also does not use any portion of the action's state α.

2.5.0.3 Race-Age

Race-Age is the most complex of the three supported race-based execution policies allowed on an action in the abstract functional interface. A Race-Age policy allows an action to effectively store the amount of work done during a previous enabling period. When a Race-Age action becomes disabled because another action has completed and the model state has changed, the action saves the amount of work done in the action state. When the action becomes re-enabled in the future, the action state is used to change the time-to-completion distribution to

reflect the fact that work has been done in the past. This action execution policy requires extensive use of the action state (α). In particular, the amount of work done before the last disabling event is saved in the **FractionComplete** member of the action state (see Section 2.4.2.7).

CHAPTER 3

IMPLEMENTING THE ABSTRACT FUNCTIONAL INTERFACE

Now that we have formally defined the abstract functional interface, we describe how we implement it in software. This chapter explains how the conceptual ideals of the abstract functional interface are realized through object-oriented programming in C++ code. The implementation of the abstract functional interface is a necessary and important prerequisite for building formalisms and solvers in the Möbius tool.

In the implementation, the abstract functional interface acts as a communication interface between the models and the solvers. For example, solvers built in the Möbius tool communicate with the model by calling methods in the abstract functional interface. These methods return generic information about the model and change the model's state. The methods that return generic information about the model can also be used to communicate information between models specified in different modeling formalisms. The implementation of the abstract functional interface thus facilitates the construction of heterogeneous models. The abstract functional interface is realized by a set of C++ abstract base classes, called the *Möbius base classes*. These classes provide the foundation for state variables, actions, groups, and models. They implement the abstract functional interface through extensive use of pure virtual methods.

The Möbius base classes provide the essential framework needed to create a tool that is easily extensible for multiple formalisms, composition methods, and solution techniques. The base classes represent the minimal subset of functionality that all models, actions, state variables, and groups must implement. The details of the implementation are specific to each formalism, but the interface is the same for all formalisms.

The use of abstraction (as implemented in the base classes) allows the formalism implementor to define the internal workings of a model in a very formalism-specific (and, one hopes, efficient) manner. For instance, the `SetState` and `CurrentState` methods in the base classes are implemented with `void` pointers to allow the formalism designer to implement a minimal model state representation in memory based upon formalism-specific information that other parts of the tool (e.g., a state-space generation module) would not know about. Thus, the tool benefits from any sort of formalism-specific shortcuts while still maintaining a common, intermodule interface.

Most of the methods in the base classes are defined as pure virtual methods since they require formalism-specific implementations. In object-oriented terminology, pure virtual functions are functions that are declared on a parent class, but are not given a specific definition. The definition is provided by the child classes, which define them in their own specific way. Because the methods on the base classes are not completely defined, they are called "abstract classes" in object-oriented terminology. Hence every formalism must realize each one of the abstract base classes to define the formalism-specific method of operation. In practice, some of the methods defined on a base class may be declared virtual instead of pure virtual, so that the method has a default definition (usually a trivial operation or an error message).

The virtual methods defined on the base classes allow different parts of the tool to access each method (e.g., `getState()`) in a formalism-specific manner. As a general rule, one incurs a performance penalty whenever virtual functions are used, because the method bindings must be determined at run-time instead of compilation-time. Nevertheless, compilers with enough knowledge of program flow can sometimes determine the derived object type for a specific virtual function call. In such instances, a call to the virtual method table is unnecessary, and will not be performed. Additional reduction in run-time overhead can be achieved by making virtual functions in-lined. Many base class virtual method implementations are trivial operations for which the majority of time needed to perform the function call is used to transfer control. This makes them ideal candidates for in-line functions [25].

The C++ classes that implement the abstract functional interface require that all necessary methods be implemented. They cannot, however, ensure that these methods are correctly

implemented for a given formalism. Validation of a formalism implementation is the implementor's responsibility.

There are four main classes that encapsulate most of the functionality of the abstract functional interface as Möbius base classes. They are `BaseStateVariableClass`, `BaseActionClass`, `BaseGroupClass`, and `BaseModelClass`. We will discuss each of these classes in the subsequent sections.

3.1 `BaseStateVariableClass`

The implementation of `BaseStateVariableClass` is relatively straightforward, because a state variable is essentially just a container for a portion of the model state. The methods defined on `BaseStateVariableClass` fall into three main categories: those that deal with the state variable's state, those that are used in state sharing, and those that are used to manipulate `BaseStateVariableClass`'s `SVAffectingActions` and `SVEnabledActions` data members. A complete list of state variable functions can be found in Table 3.1. Each set of methods will be discussed in further detail in the following sections.

State methods. Because the way state changes is specific to each formalism, `BaseStateVariableClass` has no methods defined on it for changing the state of a state variable in responsible to the firing of an action. There is, nevertheless, a pure virtual method called `SetState` defined on `BaseStateVariableClass` that changes the state of the state variable using a `void` pointer. This method is only used when the entire state of the model is being reset by a solver. Action-firing-related state changes must be implemented using derived state variable classes, implemented by a formalism designer. These methods should include a primary value function as well as any secondary value functions that may be deemed useful. The `CurrentState` method performs the inverse operation of `SetState` by writing the state variable's state to a location in memory specified by a `void` pointer input argument. The size of the state variable state can be determined by the method `StateSize`, which returns the size

Table 3.1 Methods defined on `BaseStateVariableClass`

Method Name	*Description*
`int StateSize()`	This method returns the number of bytes of compact state variable representation.
`SetName(char*)`	This method sets the name of the state variable.
`void SetState(void*)`	This method sets the state of the state variable
`void CurrentState(void*)`	This method writes the state variable's current state to the specified memory location
`void printState()`	This method prints the state of the state variable to standard out
`bool getShared()`	This returns true if the state variable is shared with another state variable
`bool getStored()`	This returns true if the state variable is using a local data member to store its state
`bool getFunctionallyShared()`	This methods returns true if the state variable value is functionally shared
`const List<BaseActionClass>* getAffectingActions()`	This method returns the affecting actions data structure
`const List<BaseActionClass>* getEnabledActions()`	This method returns the enabled actions data structure
`int getSharingCount()`	This method returns the number of state variables that are shared with this state variable
`const BaseActionClass* getAffectingAction(int)`	This method returns the specified element from the `SVAffectingActions` data member
`const BaseActionClass* getEnabledAction(int)`	This method returns the specified element from the `SVEnabledActions` data member
`int getNumAffectingActions()`	This method returns the number of affecting actions
`int getNumEnabledActions()`	This method returns the number of enabled actions
`void appendAffectingAction(BaseActionClass*)`	This method appends the specified action to the state variable's object `SVAffectingActions`
`void appendEnabledAction(BaseActionClass*)`	This method appends the specified action to the state variable's `SVEnabledActions` object
`void copyAffectingActions(List<BaseActionClass>*)`	This method copies the data structure passed in and uses it as its list of affecting actions
`void copyEnabledActions(List<BaseActionClass>*)`	This method copies the data structure passed in and uses it as its list of enabled actions
`void updateAffects(BaseStateVariableClass*)`	This method will notify all the actions on the state variable's `SVAffectingActions` and `SVEnabledActions` lists to inform them that this state variable is part of a sharing set

of the state variable state in bytes. Finally, the `printState` method is used for debugging, and prints out the name and value of all state variables in the model.

State-sharing methods. `BaseStateVariableClass` also contains state-sharing information. In particular, it contains Boolean values that indicate whether a state variable is shared (either by equivalence or functional sharing). If a state variable is shared (either by equivalence or functional sharing), then the method `getShared` will return the value "true." Thus, the `getShared` method answers the question, "Is the state variable shared?" In order to differentiate between state variables that are shared through equivalence and functional sharing, `BaseStateVariableClass` implements another method, `getFunctionallyShared`, that returns true if the state variable is functionally shared. `BaseStateVariableClass` also contains internal information used by the state-sharing code that denotes whether the state variable's state is stored locally (or points to another state variable's state). The meaning of a shared state variable state stored locally is explained later in this section. In particular, if a state variable does store its state locally, then the `BaseStateVariableClass` method `getStored` will return true. The last method in `BaseStateVariableClass` that deals with state sharing is the method `getSharingCount`. This method returns the number of state variables in the state variable's sharing set. For an unshared state variable, the method returns the value 1.

`SVAffectingActions` and `SVEnabledActions` methods. The last set of methods defined on `BaseStateVariableClass` are methods that operate on the `SVAffectingActions` and `SVEnabledActions` data structures (which are discussed in the next paragraph). The `appendAffectingAction` and `appendEnabledActions` methods allow actions to be appended to their respective data structures (by reference) while the `copyAffectingActions` and `copyEnabledActions` methods copy a set of actions pointers passed into the method.

Data members. State variables not only contain part of the model state, but also have some knowledge of the actions that affect or are enabled by their state value. This information is

encapsulated in data members (of type `List<BaseActionClass>`) that contain references to actions. The `SVAffectingActions` data structure contains all the actions whose firing could affect the state variable's state. The `SVEnabledActions` data member holds the location of all the actions that are enabled by the state variable's value. It is a good idea to implement these data members with the `List` class (see Section 3.5.1) because it handles all the necessary dynamic memory allocation and ensures that there are no duplicate actions in the list. The set of actions used to initialize these data structures for each state variable must be structurally determined from the model specification. This is analogous to the handling of data structures in `BaseActionClass` (see Section 3.2), which must be initialized with information about which state variables each action affects. No other connection information is necessary to correctly initialize the `SVAffectingActions` and `SVEnabledActions` objects if all the actions have been initialized. Figure 3.1 shows an efficient algorithm for initializing state variables based upon the connection information defined in actions.

In addition to these data members, there is a set of data members that hold information accessed through the state-sharing methods described earlier. These state-sharing methods return the values of Boolean data members that store state variable attributes. These data members include `Shared`, `Stored`, and `FunctionallyShared`. The value of `Shared` is true if the state variable is shared through an equivalence relationship and `FunctionallyShared` is true if the state variable is functionally shared. The `Stored` data member is used in constructing sharing sets, and indicates whether the shared state is stored locally. `SharingCount` is a data member (of type `int`) that stores the number of state variables in the sharing set. Lastly, `BaseStateVariableClass` contains the `StateVariableName` data member, which holds the name of the state variable.

3.1.1 Sharing through state equivalence

The most basic and lowest-level means of connecting two submodels is to have the submodels share some subset of their model states. Sharing state among homogeneous model types (models constructed within the same modeling formalism) can be done with relative ease,

```
for each a in the set of all actions
   for each s defined in a's AffectedStateVariables
      s.appendAffectingAction(a);
   end for
end for
```

Figure 3.1 Initialization algorithm for state variable `AffectingActions` objects

because both submodels are defined in terms of the same formalism-specific state variables. Therefore, each submodel understands how the other submodel's state variables are implemented. However, in a heterogeneous modeling environment in which two submodels may not be constructed using the same modeling formalism, a submodel created in one modeling formalism may not know anything about state is represented in a submodel created with another modeling formalism. In fact, this lack of knowledge about the inner workings of a model is a fundamental concept in both the Möbius framework and the abstract functional interface.

In Section 2.4.1.1 we introduced the concept of equivalence sharing. This kind of sharing is possible if we know the state variable's type. The type is the minimal knowledge we need to implement sharable state variables. In Section 2.4.1 we defined the set of all possible state variable types in the abstract functional interface. Using this definition of state variable types, we say that two state variables are "sharable" if their types are equal. This way we are assured that both state variables can take on the same values. The identity function is used as the trivial function that maps the state of the shared state variable in submodel 1 to the state of the shared state variable in submodel 2. The mapping is, by definition, one-to-one and onto. We will refer to this type of state variable sharing as *equivalence sharing*.

To understand our implementation of sharable state variables, it is important to note the distinction between a state variable and a state variable's state. In the base class implementation, a state variable is an abstract data structure that contains a portion of the model state and a set of functions that operate on that piece of state. Recall that when two state variables are shared, we do not want to create a single state variable; instead, we want to have a single variable state that the two state variables share. For instance, assume that some formalism F_1 has

a state variable class S_1 that contains a single state of type `short`. This state variable class, F_1, defines a set of functions on its state. Another formalism F_2 may define a state variable class S_2 that also contains a single state data member of type `short`, but has a different set of functions defined on it. If we decide to share these state variables, we cannot represent the shared state with a single instance of S_1 or S_2, since that would prevent one of the models from calling its formalism-specific methods on the shared state variable. Instead, we need to have both state variables operate on the same piece of state.

The simple solution to ensure that all state variables operate on the same piece of state is to have each state variable reference its piece of state by a pointer. If a state variable operates on a piece of state via a pointer, then it is easy to change the location of the shared piece of state by changing the pointers for all the state variables in the sharing set. Because shared state variables can be used again to form a new sharing relationship, it is important to be able to manage multiple sharing requests for the same state variable state.

We will now look at how sharable state variables form a sharing set and how to ensure that all state variables operate on the same state. When two state variables are shared, one state variable is declared the *leader*. The leader is responsible for updating other shared state variables if the shared state variable state location changes as a result of further sharing. The leader maintains a list of all the state variables that point to the corresponding shared state variable state. When two leaders are combined to form a single shared state variable, the larger of the two leaders (larger implying a greater number of sharers) is declared the leader of the new shared state variable state. As part of the process, the new leader annexes the other leader's list of sharers so that it can notify them of changes in the future. The result of such an operation on the two state variables shown in Figure 3.2 can be seen in Figure 3.3.

When two nonleaders are combined to form a single shared state, the larger state variable (the most number of sharers) of two associated leaders absorbs the smaller leader's list of sharers in the same way that two leader state variables are combined. The result of this type of operation can be seen in Figure 3.4. If two shared state variables that have the same number of sharers are joined together, then either state variable leader can become the leader of the new shared state variable. To avoid ambiguity, in our implementation the syntax determines which

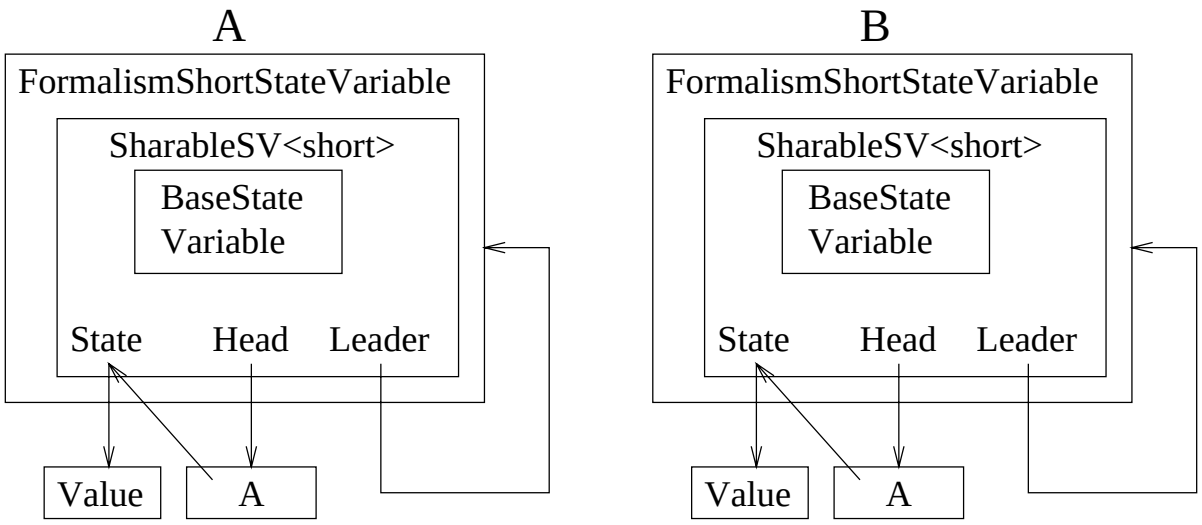


Figure 3.2 Example of two state variables not sharing state

state variable will be the new leader is both have an equal number of sharers. For example, if a state variable $A2$ is told to share state with $A1$ then $A1$ is the leader of the new sharing set.

By implementing sharable state variables in this fashion, we simplified specification of equivalence state sharing in composed models [21]. Conceptually, equivalence sharing is very simple; multiple classes operate on a single piece of memory. However, sharable state variables must update all the sharing state variables' data structures (namely `SVAffectingActions` and `SVEnabledActions`) to reflect the fact that a piece of model state is now shared across different models and thus has more actions whose firing affects the shared states. The set of affecting actions is the union of all the affecting actions for all the state variables in the sharing set.

3.1.2 Sharing through functions of state

As we saw in the previous section, it is easy to join two heterogeneous submodels through the notion of a single shared state variable. However, this notion of joining submodels requires that both models have a state variable class with the same type of state. Another way of joining heterogeneous submodels through state variables is to define the state of one state variable

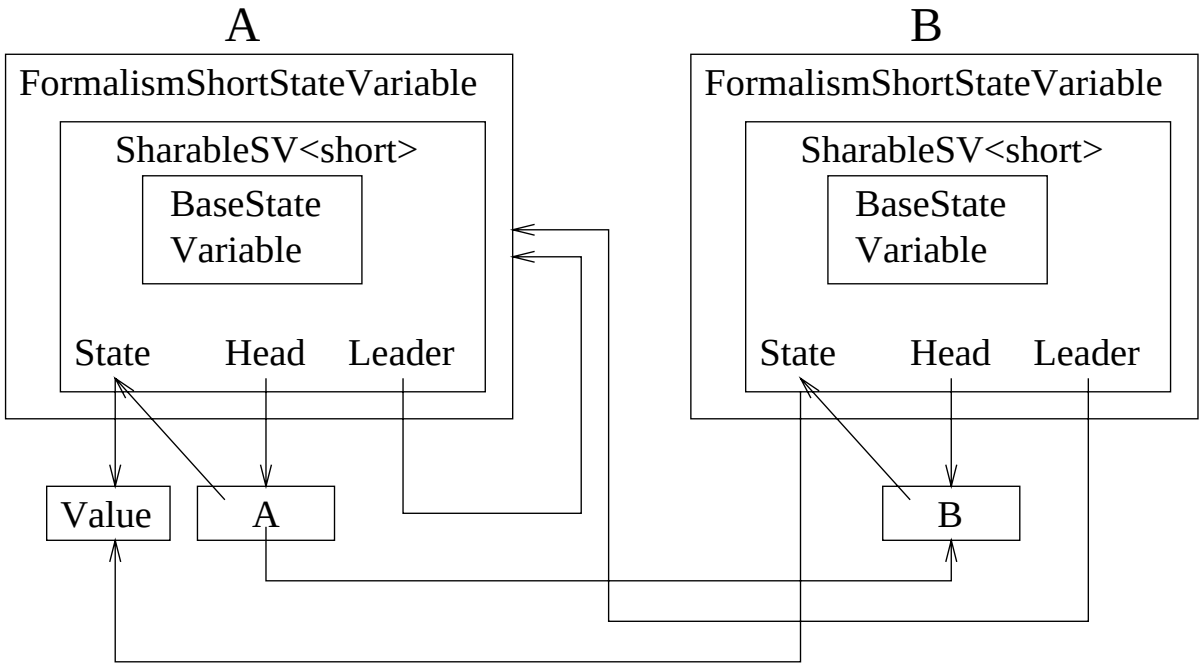


Figure 3.3 Example of two state variables sharing the same state

through a function of another submodel's state (see Section 2.4.1.1). There are two types of functional sharing: unidirectional and bidirectional.

Unidirectional functional state sharing is an inherently asymmetrical sharing relationship in which one state variable receives its value from a function defined on another state variable. The "receiving" state variable does not have any functions to set its state. This effectively makes the state variable "read-only," meaning that it can only look at the value of the state variable state, not change it. Special coding techniques are used to ensure that one cannot set the state of a read-only state variable. This is relatively easy if one allows access to a state variable only through methods. A formalism implementor can implement read-only state variables by defining a new class for which the state-changing function doesn't change the state and perhaps, instead, prints an error message. There are certain restrictions placed upon the function that defines the functional relationship between the two state variable states. In particular, the function must be surjective such that all values in the range of the sharing function must map to values in the domain of the "receiving" state variable state.

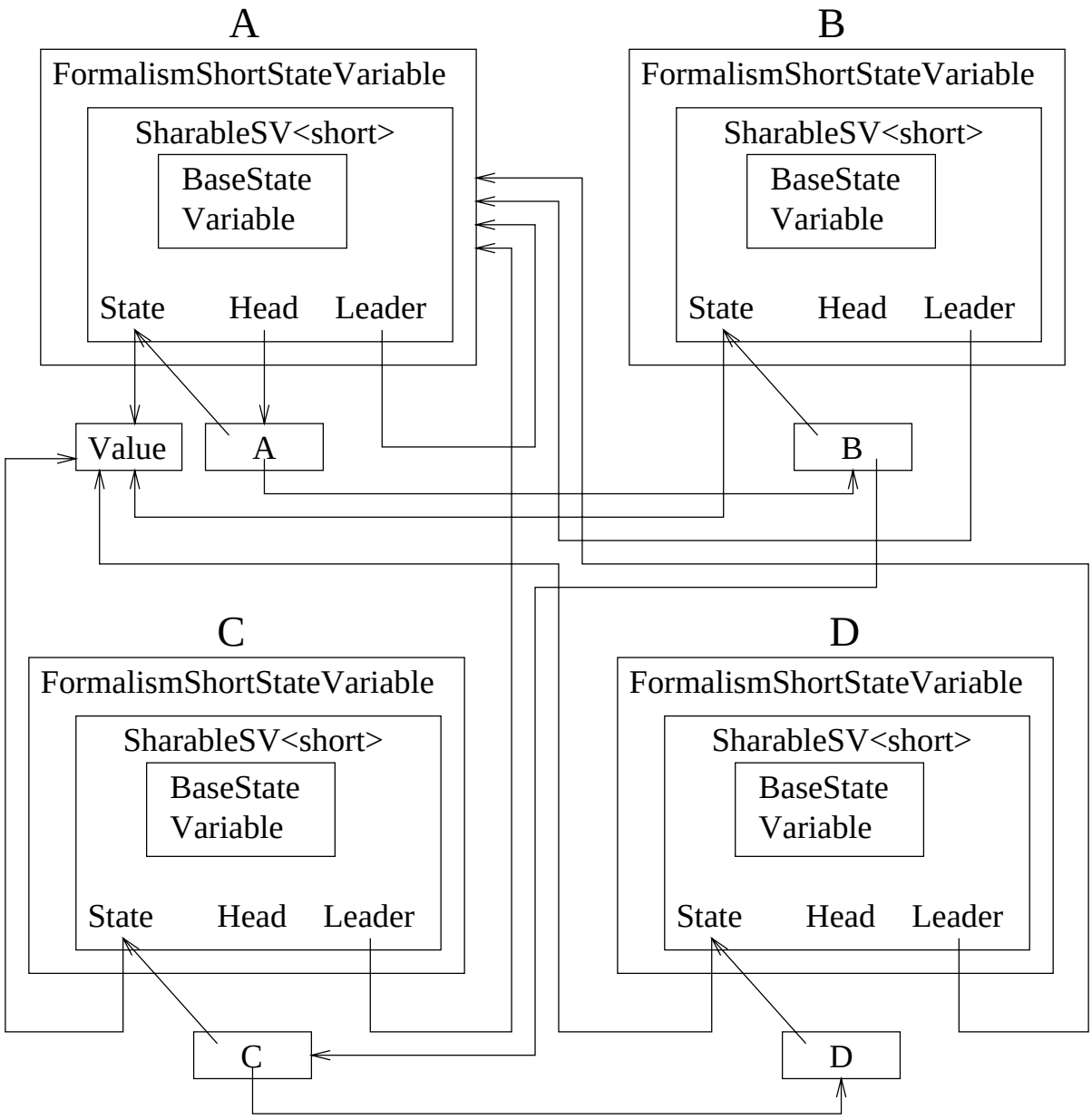


Figure 3.4 Example of two previously shared state variables of order 2 joined to form a single shared state variable of order 4

Bidirectional functional sharing removes the "read-only" condition defined on the receiving state variables. Bidirectional functionally shared state variables must be defined by a bijective function (one-to-one and onto). In addition, bidirectional functions have a clearly defined inverse function that allows the "receiving" state variable to change its state variable state. The trivial example of a bidirectional function is the identity function. We can actually use the identity function to define equivalence sharing, as a special case of bidirectional functional state sharing.

3.1.3 Implementation of the `SharableSV` class

In order to implement sharable state variables in the abstract functional interface, we must integrate state-sharing functionality into the definition of sharable state variables. In implementing a set of sharable state variable classes, we would like to have a set of classes with a common underlying functionality that ensures that all sharing state variables operate on the same state variable value, but that does not allow state variables of different types to be shared. Making a separate class derived from `BaseStateVariableClass` for each state variable type accomplishes this goal, but at the expense of having to replicate code unnecessarily. Therefore, we chose to implement the set of shared state variable classes by creating a template class called `SharableSV` (see Figure 3.5). Using this template class, one can instantiate a `SharableSV` with any one of the basic types (`char`, `int`, `float`, `double`, `short`, or `bool`). The type of the template parameter passed into the `SharableSV` template class is referenced by the name "`C`." Sharable structured, array, and unordered state variables require different implementations, since they all deal with more than one state variable value [26].

The methods declared on the `SharableSV` class (see Table 3.2) make implementation of state sharing relatively easy for models. The key method is the `ShareWith` method, which creates a sharing set that contains the `SharableSV` object on which the method is called and the `SharableSV` object passed into the method. The `ShareWith` method deallocates the memory used to store the state variable value of one of the state variables. In the process,

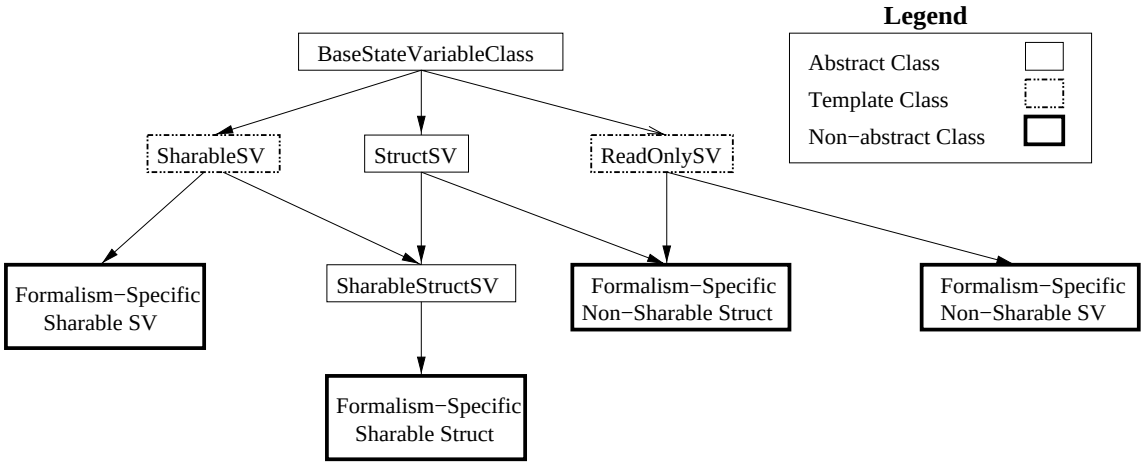


Figure 3.5 Class hierarchy for state variable classes

the `ShareWith` method forces the state variables in the sharing set to operate on the same shared state variable value. The state variable passed into the `ShareWith` method is also added to the linked list of sharers maintained by the sharing set leader. (This is done by calling another method on `SharableSV` called `Register`.) Maintaining a linked list of sharers is very important in maintaining a sharing set. If the location of the shared state variable value is moved, then the linked list is used to update all the pointers used to reference the state variable value.

There are two implementations of the `Register` method. One takes an argument of type `SharableSV`, and the other takes an object of type `C**`. The latter is used to register the address of a "shortcut pointer." *Shortcut pointers* are pointers that point directly to the value of the state variable. Shortcut pointers are useful in implementing models in which state changes must be done quickly. Actions that operate on state usually incur at least two pointer dereferences when calling a state variable value function. The first dereference occurs when the action dereferences the pointer to the state variable (state variables are declared in the model, not in actions); the second dereference occurs when the state variable dereferences its pointer to state. However, shortcut pointers allow a model to manipulate a state variable's value outside of the state variable value function. If a state variable has a complicated value function, then reimplementing this code elsewhere in the model with shortcut pointers increases code size

Table 3.2 Methods defined on `SharableSV` class

Method Name	*Description*
`void ShareWith(SharableSV<C>*)`	Implements equivalence sharing with another state variable of type `SharableSV<C>`
`void Register(SharableSV<C>*)`	Builds a list of pointers pointing to shared memory location
`void Register(C**)`	Registers a shortcut pointer to state variable state
`int StateSize()`	Returns the size of the sharable state variable in bytes
`void setPointer(C*)`	Sets the location of the shared state
`void SetState(void*)`	Sets the value of the shared state variable state
`void CurrentState( void*)`	Copies the value of the shared state variable to a location in memory
`void setLeader(SharableSV<C>*)`	Sets the leader of the sharing set
`SharableSV<C>* getLeader()`	Returns the leader of the sharing set
`SharedPointer<C>* getHead()`	Returns the head of the linked list of shared state variables and shortcut pointer
`C getState()`	Returns the value of the shared state variable
`C& State()`	Returns the value of the shared state as an lvalue
`void setState(C)`	Sets the value of the shared state variables
`void appendAffectingAction( BaseActionClass*)`	Appends an affecting action to all the state variables in the sharing set

and the risk of implementing the state change incorrectly. As a general rule, shortcut pointers should only be used when the state changes are relatively simple (e.g., incrementing a value by one, adding a constant value).

Shared state variables must implement their `appendAffectingAction` and `appendEnabledSV` methods differently from non-shared state variables, since a shared state variable can be affected by actions that are not in the same atomic model. In this case, the `appendAffectingAction` and `appendEnabledSV` methods operate on all members of the sharing set. When a new state variable is added to the sharing set, the `ShareWith` method ensures

that every state variable has the same affecting actions. This is done by building a master list of affecting actions by looking at each member of the sharing set's list of affecting actions.

Since `SharableSV` is not an abstract C++ class, it implements pure virtual methods as defined in `BaseStateVariableClass`. These methods include `StateSize`, `SetState`, and `CurrentState`. Both the `SetState` and `CurrentState` methods make use of template functions used to read and write values to memory across word boundaries (see Section 3.6). These template functions determine the size of the state variable value by using the `sizeof` function, and write the state variable value one byte at a time. Having the ability to write across word boundaries increases our memory efficiency when we are saving the state of a state variable or model.

Table 3.3 lists `SharableSV` data members. The `TheState` pointer is the `SharableSV`'s pointer to the state variable's value. The `Leader` pointer points to the sharing set's "leader." In a sharing set, the *leader* state variable is the state variable that contains the value of the shared state variable. The leader is also entrusted with the responsibility of maintaining the sharing set's data structures. Lastly, the `Head` data member points to the head of a linked list that contains references for all the members in the sharing set. This linked list is important for implementing any method that affects all members of the group.

Table 3.3 `SharableSV` data members

Data Member	*Description*
`C* TheState`	Where the state is currently located
`SharableSV<C>* Leader`	Points to the state variable holding the shared piece of state
`SharedPointer<C>* Head`	Points to a linked list of other state variables sharing the same state

3.1.4 Unidirectional functionally sharable state variables

Unidirectionally functionally shared state variables are also implemented by a base template class. The base class for functionally shared state variables is called `ReadOnlySV` (See Table 3.4). This class contains a function pointer, `(C *getValue)()`, which is used to

reference the function that defines the functionally shared state variable's value. This function pointer is initialized with the method `setFunction(C (), List <BaseStateVariableClass>)`, which is defined in `ReadOnlySV`. This method takes a function pointer parameter that is used to initialize the `getValue` data member and a list of state variables. The list of state variables corresponds to the state variables used in the value function definition. It is important to have these state variables so that the functionally shared state variables maintain a correct list of affecting actions. The affecting actions list for a functionally shared state variable is constructed by forming the union of all the affecting actions for all the state variables used in the value function definition. Therefore, whenever the `setFunction` method is called on a `ReadOnlySV`, the state variable's `SVAffectingActions` data structure must be updated. Because the unidirectionally functional sharing is a one-way relationship, the `ReadOnlySV` class does not need to update the `SVEnablingActions` data structure, since all of the state variable's enabling actions should be defined within the same model that the `ReadOnlySV` state variable is defined in.

Because functional sharing relationships are specified as part of composed model specification, the composed model is responsible for initializing the `getValue` function pointer for all functionally shared state variables. The user defines the value function used for functional sharing as part of the composed model specification. It is therefore the composed model's responsibility to create an executable function that can be bound to a `ReadOnlySV`'s function pointer as well as to determine the set of state variables used in the function definition; both the function and the state variables are needed by the `setFunction` method.

Table 3.4 Methods defined on `ReadOnlySV` template class

Method Name	*Description*
`int StateSize()`	Overloaded virtual method returns 0
`void SetState( void*)`	Overloaded virtual method does nothing, because functionally shared state cannot be set
`void setFunction( C (*TheStateFunction)())`	This method sets the state variable's function pointer. The function pointer will be used to determine the state variable's value

Figure 3.6 shows how the concepts of functional and equivalence sharing can be used to compose models expressed in different modeling formalisms. One atomic model is expressed in the stochastic activity network formalism and contains `Place` state variables, each of which has a single value function (`Mark`) of type `short`. The queuing network atomic model contains two types of state variables: `MultiClassQueue` and `SingleClassQueue`. `MultiClassQueue` is a structured state variable that has three secondary value functions: `getID` (of type `int`), `getColor` (of type `char`), and `getQueueLength` (of type `short`). The last atomic model is expressed in the Petri Net atomic model formalism, and has `Place` state variables, each of which has a single value function, `getMark`, of type `short`. Figure 3.7 shows how equivalence and functional sharing relationships can be declared using these three atomic models.

3.2 `BaseActionClass`

The description of `BaseActionClass` is straightforward given that the necessary functions have already been identified and described in Section 2.4.2. In addition to the action functions defined in this section, is also a small set of utility methods defined on `BaseActionClass`. Since the base action functions have already been described in great detail, this section will describe the remaining utility method implementations and assorted data structures defined on `BaseActionClass`. Table 3.5 is a complete list of methods defined on `BaseActionClass`.

Implementing `AffectedStateVariables` and `EnablingStateVariables`. The abstract functional interface requires that all actions define a set of state variables whose value enables the action and whose value is affected by the action's firing. We formally defined these sets when we defined the abstract functional interface in Section 2.4.2.1 and Section 2.4.2.2. The data structures named `EnablingStateVariables` and `AffectedStateVariables` represent the sets S_{a_ϵ} and S_{a_ϕ} respectively (see Section 2.4.2.1 and 2.4.2.2). These data structures allow for many efficiencies during model solution. For example, both sets of

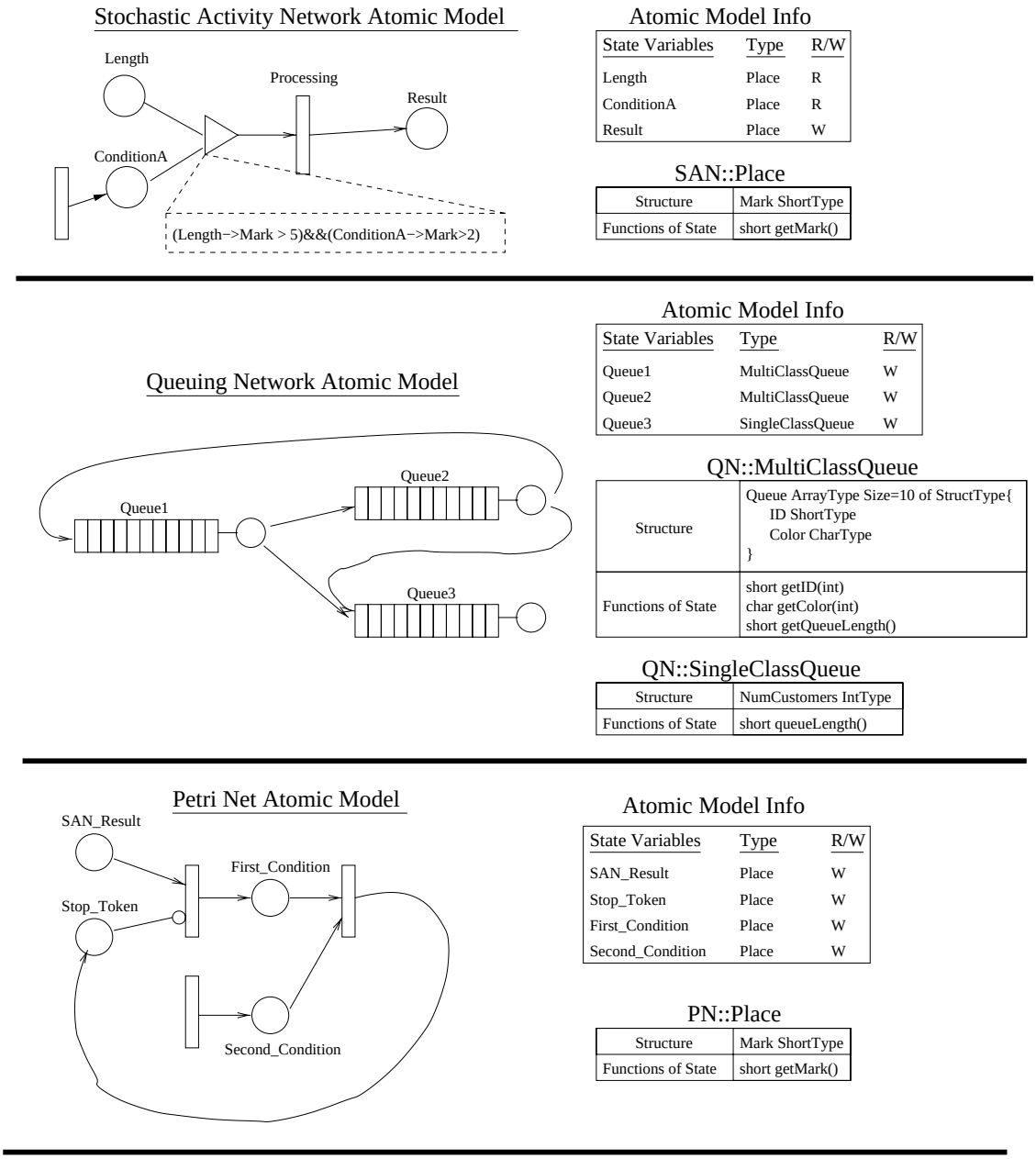


Figure 3.6 Three different formalisms with functionally-sharable state variables

Table 3.5 Methods defined on `BaseActionClass` that implement abstract functional interface action functions

Method Name	*Description*
`bool Enabled()`	This method determines whether the action is enabled in the current state
`double Weight()`	Weights are used to determine the probability of selecting an action from the set of enabled actions in the current state
`double Rate()`	This method returns the rate with which an exponentially timed action fires
`bool ReactivationPredicate()`	This method determines whether an action is reactivatable
`bool ReactivationFunction()`	This method determines whether an action whose `ReactivationPredicate` is true should restart after a state change in which the action is still enabled
`double SampleDistribution()`	This method samples the action's distribution and returns the action's time-to-completion
`double* ReturnDistributionParameters()`	This method returns the set of distribution parameters
`void SetFired()`	This method sets the `Fired` data member on an action to record the fact that the action fired
`BaseActionClass* Fire()`	This method defines how the action changes the state of the model
`int Rank()`	This method returns the action's priority value for a given state
`bool EnablingChange()`	This method determines whether there has been a change in the enabling condition since the last time the `Enabled` method was called
`bool IsAMember( BaseActionClass* TheAction)`	This returns true if the specified action is equal to the `this` object
`double Probability( BaseActionClass* TheAction)`	This method returns 1.0 if the specified action is equal to the `this` object, or 0 otherwise

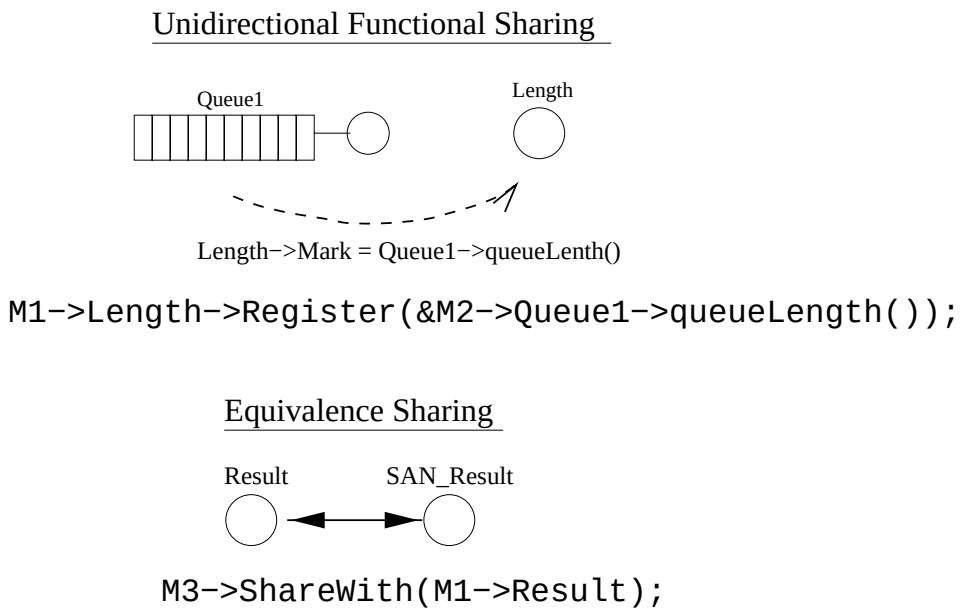


Figure 3.7 An example of equivalence and functional state sharing

state variables are used to increase the efficiency of calculating reward values and discrete-event simulation. We specify both sets of state variables for all actions in the model so we can determine which actions' state changes could affect the enabling status of other actions. Therefore, after every state change, we only have to inspect the enabling status of a subset of the actions.

The definitions of the action attributes EnablingStateVariables and AffectedStateVariables (see Section 2.4.2.1 and Section 2.4.2.2) are based upon a model's behavior, because they require knowledge about which states can be reached from other model states. Because this knowledge is often not known at model specification time, we must use structural information to deduce a superset of S_{a_ϵ} and S_{a_ϕ}. Because `EnablingStateVariables` and `AffectedStateVariables` data structures are only used for the sake of efficiency, having more than the necessary number of members in `EnablingStateVariables` and `AffectedStateVariables` will not affect the correctness of the model solution. If we cannot deduce anything about which state variables are enabling or affected by an action, then we must put all the model's state variables in `EnablingStateVariables` and `AffectedStateVariables` data structures.

Both the `EnablingStateVariables` and `AffectedStateVariables` data structures are static containers for state variables. The set membership of these two data structures should not change after model initialization. For these reasons, both `EnablingStateVariables` and `AffectedStateVariables` were implemented as instances of the `List` template class. Both of these data structures were declared as `protected` members of `BaseActionClass` to prevent models and solvers from changing their values during model execution. `BaseActionClass` defines a set of simple functions that return all the important information about these data structures. Table 3.6 lists all of these methods.

There are two methods for each data member that change the value of the data structures. The `addEnablingSV(BaseStateVariableClass*)` and `addAffectedSV(BaseStateVariableClass*)` methods should be used at model initialization time to define the corresponding set's members. State variables passed into these methods will be appended to the data structures if they are not already members. If a state variable is already contained in the `List` then it will not be added again. The `replaceAffectedSV(BaseStateVariableClass*, BaseStateVariableClass*)` and `replaceEnablingSV(BaseStateVariableClass*, BaseStateVariableClass*)` are methods used when a state variable is shared among multiple models. In order to determine which actions affect which other actions in a composed model, members of the `EnablingStateVariables` and `AffectedStateVariables` must always point to the leader of the sharing set (see Section 3.1.1).

Action data structures. Actions have many other data structures in addition to the `EnablingStateVariables` and `AffectedStateVariables`. Some of these data structures represent various action attributes: others are solver-specific data structures used for executional efficiency.

We will first look at the data structures used to represent action attributes. A list of all of these action attribute data structures can be found in Table 3.7. The `GroupID` action attribute indicates to which group the action belongs. In the case of multilevel groups, an action's `GroupID` corresponds to the highest-level group to which the group belongs. The highest-

Table 3.6 `BaseActionClass` methods used to access information about `AffectedStateVariables` and `EnablingStateVariables` data structures

Method Name	*Description*
`int getNumEnablingSVs()`	Return the number of enabling state variables
`void replaceEnablingSV( BaseStateVariableClass*, BaseStateVariableClass*)`	This method allows a model to replace an element of the `EnablingStateVariables` with a new state variable location
`BaseStateVariableClass* getEnablingSV(int)`	This method returns the *i*th member of the `EnablingStateVariable` data member
`void addEnablingSV( BaseStateVariableClass*)`	This method adds an enabling state variable to `EnablingStateVariables`
`int getNumAffectedSVs()`	Returns the number of enabling state variables
`void replaceAffectedSV( BaseStateVariableClass*, BaseStateVariableClass*)`	This method allows a model to replace an element of the `AffectedStateVariables` with a new state variable location
`BaseStateVariableClass* getAffectedSV(int)`	This method returns the *i*th member of the `AffectedStateVariable` data member
`void addAffectedSV( BaseStateVariableClass*)`	This method adds an enabling state variable to `AffectedStateVariables`

level group corresponds to the set of groups that logically partitions all the actions in a model. The `GroupID` is used to identify which actions belong to a specific group. For instance, one could identify all the actions belonging to the group of instantaneous actions by checking whether the `GroupID` is equal to zero (all instantaneous actions belong to group zero).

The `ExecutionPolicy` data member represents the action's execution policy type. As stated in Chapter 2, the Möbius abstract functional interface supports three different race-based action execution policy types: Race-Resampling (Section 2.5.0.1), Race-Enabled (Section 2.5.0.2), and Race-Age (Section 2.5.0.3). The `ExecutionPolicy` data member is stored as an enumerated type `ExecutionPolicyType` that has values `RaceResampling`, `RaceEnabled`, and `RaceAge`. This action attribute is used to tell solvers how to deal with actions that become disabled before they fire.

In addition, BaseActionClass includes another action attribution DistributionType, which defines the type of probability distribution used for describing the action's firing time distribution. This distribution type can be one of those supported by the Möbius abstract functional interface (a complete listing which can be found in Table 3.8 with parameters defined in [27]), or one specified by the model. The DistributionType attribute is also saved as an enumerated type with one value labeled Proprietary when the distribution type is not one of the supported distributions. If the modeler does not want to use one of the predefined distribution types, the user-defined distribution function must be completely specified within the double* SampleDistribution() method such that the method correctly returns a sample from the distribution. Lastly, every action has a Name attribute that uniquely identifies it within the model.

Table 3.7 BaseActionClass attributes data structures

Attribute Name	*Description*
int GroupID	The highest-level group to which the action belongs.
ExecutionPolicyType ExecutionPolicy	The type of race-based execution policy that should be applied to the action.
char* ActionName	The name of the action.
Distribution DistributionType	The type of distribution function used for the action's firing time distribution.

Solver-specific data structures. There are a number of data structures defined on BaseActionClass used only by solvers (see Table 3.9). Having these data structures defined on abstract functional interface base classes simplifies the implementation of solvers without interfering with action functions. The first data structure that falls into this category is the Affects data member. This data member is the head of a linked list created by a solver that contains information about which actions are affected by the action's firing. One action affects another action by having a Fire method that changes the state of the second action's enabling state variables. A solver can create a list of actions affected by one action's firing by using BaseActionClass's data structures EnablingStateVariables and AffectedStateVariables. The data structure EventPointer can be used by simulation-based solvers

Table 3.8 Supported distribution functions in `UserDistributions` class

Distribution Name	*Parameters*
`Exponential`	Rate
`Deterministic`	Value
`Geometric`	p
`Weibull`	α, β
`Normal`	μ, σ^2
`Lognormal`	μ, α^2
`Erlang`	m, β
`Triangular`	a, b, c
`Gamma`	α, β
`Beta`	α1, β1
`Uniform`	UpperBound, LowerBound
`Binomial`	t, p
`NegativeBinomial`	s, p
`HyperExponential`	rate1, rate2, p

to keep track of events created by an action's firing. Because different solvers may implement different event data structures, the `EventPointer` is maintained as a `void` pointer. This allows a simulator to cast the `EventPointer` to whatever event data structure it uses. Lastly, `BaseActionClass` maintains a pointer to the atomic model to which it belongs.

Table 3.9 `BaseActionClass` data structures used by simulators

Data Structure	*Description*
`ActionAffectsElement* Affects`	The head of a linked list of other actions that are affected by this action's firing. The solver creates this linked list.
`void* EventPointer`	A pointer to an action's associated event on a simulator future event list.
`void* TheModel`	A pointer to the atomic model that contains the action.

`BaseActionClass` state data structures. Another set of data structures on `BaseActionClass` stores action state for the associated action (see Table 3.10). The enabling status of an action in a previous state is stored in the `Disabled` data structure. This data structure

is used by groups when they choose their representative members. In the case of variable preselection, the enabling status of all the group members at selection time must be recorded such that new representative members can be chosen in future states. A variable preselection group can then compare the value of each member's `Enabled()` function against the `Disabled` value saved in the action state. If the enabling status has changed for any of the group members, the variable preselection group knows that it must reselect a new representative action.

Another data member in `BaseActionClass`, a value called `Reactivation`, saves the value of the reactivation predicate that was evaluated when the action became enabled. This part of action state must be saved so that future states can correctly determine when an action should be restarted.

The last portion of action state stored as a data member in `BaseActionClass` is `FractionComplete`. `FractionComplete` is stored as a `double` value and should always have a value in the range [0,1). `FractionComplete` represents the amount of work done by the action in all previous states since its last firing. This value is used by solvers to scale the amount of time the action is enabled before it fires in future enabling states.

Table 3.10 Action state data structures

Data Structure	*Description*
`bool Disabled`	This Boolean flag is set when the enabling status of an action in a group is being evaluated
`bool Reactivation`	The value of the reactivation predicate at enabling time
`double FractionComplete`	The fraction of work done before the action last became disabled

Performance variable related data structures. `BaseActionClass` also contains several different data structures that are used to implement performance reward variables efficiently. Performance variables define a set of "workers" on an action. A *worker* is an object that updates a specific reward variable in response to the firing of a specific action. It is thus the responsibility of the performance variable model to define these workers according to which state variables and actions are used to define the reward. Because these data structures are

specific to particular types of reward models, they should not be regarded as parts of the abstract functional interface. A formalism implementor does not need to do anything special with these data structures, since they are managed entirely by the performance variable reward model. For completeness, all of the performance-variable-specific data members are listed in Table 3.11.

3.3 `BaseGroupClass`

The C++ class `BaseGroupClass` implements the group functions highlighted in Section 2.4.3. `BaseGroupClass` implements a default implementation of the methods where it is applicable, but most methods are declared as virtual functions to allow formalism implementors to create new group implementations (see Table 3.12 for a complete listing of group functions). `BaseGroupClass` is derived from `BaseActionClass` and thus has many of the same methods as `BaseActionClass`. The implementation of these `BaseActionClass` methods is defined by the specific type of group involved.

`BaseGroupClass` provides two methods for constructing the group member list: `appendGroup` and `appendMembers`. The `appendGroup` method makes the group passed into the method a member of the group on which the method is called. The `appendMembers` method adds the members of the group passed into the method to the object on which `appendMembers` is called. `BaseGroupClass` maintains its membership list in two protected data members: `GroupMembers` and `ActionMembers`. Both of these data members are of type `List` and hold pointers to their respective base class types (either `BaseActionClass` or `BaseGroupClass`).

The real functionality of a group is embedded in the methods that select a representative action from the group members. The `SelectAction` method performs the actual selection of a group member. The default implementation for `SelectAction` in `BaseGroupClass` uses members' rank and weight values to determine the selected action. This selection algorithm is outlined in Figure 3.8. The selection algorithm starts out by creating a discrete probability distribution function based upon the all of the group members in the highest-ranked group. The protected method `CalculateWeightDistribution` generates this distribution function

Table 3.11 Performance variable action data structures

Data Structure	*Description*
`ActionAffectsElement* Affects`	Linked list of state variables affected by the firing of action
`int* PVAffects`	The list of performance variables whose reward functions are affected by the action
`int NumPVImpulseAffects`	The length of the `PVImpulseAffects` array
`int* PVImpulseAffects`	A list of performance variables whose impluses are affected by this action
`int** PVImpulseAffectsImpulses`	The list of impulses on the affected performance variables
`int*** PVImpulseAffectsImpulseWorkers`	The list of workers defined on the impulse-affecting impulses (from `PVImpulseAffectsImpulses`)
`int* NumPVImpulseAffectsImpulses`	The number of the impulse workers array in `PVImpulseAffectsImpulse`
`int** NumPVImpulseAffectsImpulseWorkers`	The length of the impulse workers array in `PVImpulseAffectsImpulseWorkers`
`int* NumPVWorkers`	The number of `PVWorkers` defined on each performance variable
`int **PVWorkerList`	An array of `PVWorker` arrays. A set of `PVWorkers` is defined upon each performance variable.
`int TotalNumCollected`	The total number of performance variables collected to date
`int TotalNumAffects`	The length of the `TotalNumAffectsList`
`int* TotalPVAffects`	A complete list of performance variables affected by this action
`int NumAffects`	The length of the `PVAffectsList`

Table 3.12 Methods defined on `BaseGroupClass`

Method Name	*Description*
`void appendGroup(BaseGroupClass*)`	This method adds the specified group to the list of member groups
`void appendMembers(BaseGroupClass*)`	This method adds the specified group's members to this group
`void SelectAction()`	This method performs the selection algorithm on the group and defines which of the group's actions is selected
`double CalculateWeightDistribution()`	This method is used to calculate the probability of selecting each member action in the current state
`double Probability(BaseActionClass*)`	This method returns the probability of selecting the specified member action from among the set of enabled member actions in the current state
`bool IsAMember(BaseActionClass*)`	This method checks to see whether the specified action is a member of the action group
`int getNumMembers()`	This method returns the number of group members
`int getNumGroupMembers()`	This method returns the number of group members that are groups
`int getNumActionsMembers()`	This method returns the number of group members that are actions
`BaseActionClass* getSelectedAction()`	This method returns the action selected by the group
`BaseGroupClass* getGroupMember(int)`	This method returns the *i*th member that is a group
`BaseActionClass* getActionMember(int)`	This method returns the *i*th member that is an action.
`void printGroup()`	This method hierarchtically prints out a group's membership

by creating a linked list, in which each element represents a discontinuity in the distribution function. The algorithm for building the distribution function is outlined in Figure 3.9.

The last method that deals with member selection is the `Probability` method. This method is necessary for many analytical/numerical solvers, because it returns the probability of a certain action firing in the current state. If a group uses the default selection algorithm, then the probability of selecting a member is the ratio of the member's weight value and the sum all enabled highest-ranked members. Many times a group member may itself be a group. In such cases the `Probability` method is called recursively. The final probability of selecting an action is the probability of selecting each group to which the action belongs times the probability of selecting the action from the "lowest-level" group.

3.4 `BaseModelClass`

`BaseModelClass` defines the abstract functional interface for all models in Möbius. One of the main functions of a model in Möbius is to provide solvers and other models with access to Möbius entities within a model (e.g., state variables, actions, groups, other models). A summary of these methods is provided in Table 3.13. Methods that provide structural information about the model are often used by both models and solvers. This structural information is usually contained in a set of base class objects returned by the methods. These base class objects form the basis for many inter-model operations (like model composition and connection) and solution methods. For instance, solvers only need to deal with actions and groups, because solvers are only concerned with firing successive sets of actions to change the model state. Composed models that use a notion of equivalence sharing do not need to know anything about submodels other than what state variables they contain.

We can divide the methods defined on `BaseModelClass` into three categories: the list methods, the state methods, and the composed model methods.

List methods. The list methods we defined on `BaseModelClass` return a set of base class objects (`BaseModelClass`, `BaseGroupClass`, `BaseActionClass`, or `BaseState-`

```
generate distribution function as a linked list
generate random number
weight value = head of linked list
while random number < weight value
   weight value = next value in linked list
end while
SelectedMember = member corresponding to current linked list element
if selected member is a group
   SelectedAction = SelectedMember.SelectAction()
else
   SelectedAction = SelectedMember
```

Figure 3.8 Algorithm for selecting a group member

```
higest ranked member = -1;

while there are still unchecked members
   if Member is Enabled
      if Member.Rank == highest ranked member AND Member.Enabled()
         next element in linked list = last element + Member.Weight()
      else if Member.Rank() > highest ranked member AND Member.Enabled()
         highest ranked member = Member.Rank()
         reset linked list to head
         make first linked list element = Member.Weight()
      else
         get next member
end while

if highest ranked member == -1
   SelectedAction == NULL
else
   divide each element in linked list by value of last element
```

Figure 3.9 Algorithm for calculation of member distribution function

Table 3.13 Methods defined on `BaseModelClass`

Method Name	*Description*
`void listModels(char*, List <BaseModelClass>*)`	The function returns a list of references to all the other models with the specified name defined within a model, including itself
`void listActions(List <BaseActionClass>*)` hline `void listActions(char*, List <BaseActionClass>*`	This returns a reference to all of the actions contained in a model This method returns all the actions contained in the model with the specified name
`void listGroups(List <BaseGroupClass>*)`	This returns a reference to all of the action groups contained in a model
`int getNumActions()`	This returns the number of actions in a model
`int getNumGroups()`	This method returns the number of groups contained in the model
`int StateSize()`	This function returns the size of the memory needed to save the model's current state
`bool CompareState(void*, void*)`	This function compares two model state representations and determines whether the two representations are the same model state
`void listSVs(char*, char*, List <BaseStateVariableClass>*, List <BaseModelClass>*)`	This method returns a list of references to state variables that have a specific name in a specific model (as specified by the caller)
`int CountAffectedVars(char*, char*)`	This method returns the number of state variables with a specific name and in a specific model
`void CurrentState(void*, void*)`	This method writes the model's current state to a specified memory location
`BaseStateVariableClass* getMainSharedVariable( BaseStateVariableClass*)`	This method hierarchically determines the highest-level state variable that the state variable has been shared with through the composer tree
`void printState()`	This method prints the state of the model to stdout. It is used for debugging purposes
`SharedStateVarLink* getListOfSharedVariables( BaseStateVariableClass*)`	This method is used to hierarchically build groups of equivalent state variables shared at each level in the composer tree
`updateAffectsList( BaseStateVariableClass*, BaseStateVariableClass*)`	This method changes the data structures of all actions in the model such that the actions use a new location for a specified state variable
`void SetState(void*)`	This method sets the state of the model

`VariableClass`) to the other models or solvers. All of these methods make use of a template class called `List` (see Section 3.5.1 for implementation details), which provides simple methods to append and retrieve members. We implemented the list methods such that a pointer to a `List` object is passed into the list method. The method then uses the pointer to "fill up" the list with the appropriate abstract functional interface entities. Since solvable models are always made up of several smaller models, list method implementations often require calling the same method on lower-level models. The `List` pointer is thus passed down to the lower-level models, which continue to fill the `List` object with the appropriate abstract functional interface entities. This paradigm of creating a single object that is passed down to lower-level models is much more effective than having each model dynamically create its own list.

Some list methods are "global" in the sense that they return all the entities contained in a model of a certain type. These global list methods are implemented by the methods `listActions(List<BaseActionClass>*)`, `listSVs(List<BaseGroupClass>*)`, and `listGroups(List<BaseGroupClass>*)`. Another category of list methods is that of filters that return a subset of abstract functional interface entities. These are usually name filters that return the set of entities with some specified name. Because each model has a separate name space, it is possible to have an executable model that contains many lower-level models with identically named entities. This type of information is valuable for specifying reward structures. The `BaseModelClass` list methods that use a name filter are `listModels(char*, List<BaseModelClass>*)`, `listActions(char*, List<BaseActionClass>*)`, and `listSVs(char*, char*, List<BaseStateVariableClass>*, List<BaseModelClass>*)`. The last of these three methods returns a subset of state variables that have the specified name. Both that state variable name and the model name are given as input parameters. The `listSVs` method returns pointers to the state variables that meet these criteria and pointers to the models in which each of these state variables exists. Thus, the state variable list and the model list should contain the same number of elements. The list indices are correlated such that the ith element in the state variable list corresponds to the ith item of the model list (where i is a valid index into the list).

The state methods. `BaseModelClass` also defines a set of methods that deal with the model's state variable state. We intentionally implemented these methods as abstractly as possible, using `void` pointers as parameters. By doing so, we avoided making requirements about the format of state being passed in and out of these `BaseModelClass` methods. It is understood that formalism implementations of `BaseModelClass` will know the format of the model state and cast model state accordingly. Thus, Möbius components that manipulate state variable state (namely the solvers) deal with pieces of memory. The methods that fall into this category include `SetState(void*)`, `CurrentState(void*)`, and `CompareState(void*, void*)`. The `SetState` method sets the values of a model's state variables using the memory located at the specified address. `CurrentState` performs the inverse action, by writing the model's state variable state to the specified location in memory.

Lastly, the `CompareState` method is used to compare two model states located in memory. It is not correct to say that two model states are not equivalent if the values in memory are different. In order to compare state, one *must* use the `CompareState` method defined on `BaseModelClass`. One reason is that a formalism may use a very simplistic way of storing state, but have a sophisticated implementation of `CompareState` that detects symmetries in the state space and creates sharing sets of state variable state. There is one other method that deals with state variable state in a model `StateSize()`, which returns the state size in bytes.

Composed model methods. `BaseModelClass` also provides two methods that return information that is necessary for building composed models through a notation of shared state and another method that is necessary for implementing composed models. Some composed model formalisms use a notion of shared state to join two models together structurally. For that type of composed model formalisms, we say that the two state variables joined together are members of the same sharing set (defined in Section 2.4.1.1). Every sharing set has a single state variable that is declared the sharing set's leader (see Section 3.1.3).

The first composed model method defined on `BaseModelClass` returns a sharing set's leader. The method `getMainSharedVariable(BaseStateVariableClass*)` re-

turns a reference to the sharing set's leader. Another method defined on `BaseModelClass` used in model composition is `getListOfSharedVariables(BaseStateVariableClass*)`, which returns a pointer to the head of a linked list of all the state variables in the sharing set. The last method, `updateAffectsList(BaseStateVariableClass*, BaseStateVariableClass*)`, is used to update action data structures. Actions contain data structures with state variable pointers. It is important that some of these state variable pointers point to the leader of the state variable's sharing set. The `updateAffectsList` method changes these data structures by replacing all instances of the shared state variable pointer with the location of the state variable's sharing set leader.

3.4.0.1 Model data structures

Every model has a set of public variables that define base model attributes. Most of these data structures summarize the quantities of each of the different abstract functional interface entities in the model. A complete listing in given in Table 3.14.

Table 3.14 Data structures defined on `BaseModelClass`

Data Structure	*Description*
`int NumStateVariables`	The number of state variables in the model
`int NumSharedStateVariables`	The number of state variables that are shared through equivalence sharing
`int NumActions`	The number of actions in the model
`int NumGroups`	The number of groups in the model
`int NumPVs`	The number of performance variables in the model
`char* Name`	The name of the model
`BaseGroupClass** GroupList`	The list of all groups in the model

3.5 Additional Data Structures

Implementing the abstract functional interface is aided by additional classes that help communicate specific types of information that cannot be encapsulated in one of the four main

base classes. In order to provide a complete description of the abstract functional interface implementation, a description of these additional classes is provided here.

3.5.1 `List`

Many methods in the abstract functional interface require a set of Möbius entities as input parameters or as a return type. Many of these methods require that these sets be created dynamically. In order to minimize the amount of code involved in the process of allocating and deallocating memory, the abstract functional interface uses a paradigm in which functions that require a set of entities as a return type pass an empty "container" to the function. The function then fills up the container with the required entities. This paradigm involves removing dynamic memory allocations, as well as locally declared variables used as return values, from method implementations. In the case of recursive method implementation, this container object is passed down to lower-level method calls. This also happens in functions defined on higher-level models that call the the function with the same name on contained submodels; each contained model adds its information to this common object.

Because container classes are required for all Möbius entities and differ only in the types of objects stored in them, we decided to implement the container class as a template class called `List`. The `List` class manages a protected data member, which is an array of pointers to Möbius entities. Each container class implements a method to append a single Möbius entity, an array of entities, or another instance of the same `List` class (see Table 3.15). Appending another instance of the class implies that all the entities in the other instance will be appended to the local list. The default implementation does not allow the same entity to appear on the list more than once. An example use of the `List` class is the `AffectingActions` object defined on `BaseStateVariableClass`.

Dynamic memory allocations are handled efficiently by having the `List` class allocate a large piece of memory at construction time. This reduces the probability of having to allocate more space for the list in the future due to an excessive number of entities being appended to the list.

Table 3.15 Methods defined on `List<class C>` template class

Method Name	*Description*
`const int getNumItems()`	Returns the size of the list
`const C** getList()`	Returns a copy of the list
`void append(List*)`	Appends another list's elements (removing duplicates)
`void append(C* )`	Appends an item to the list
`void append(C**, int)`	Appends an array of items to the list
`C* getItem(int)`	Returns the *i*th element from the list
`void replaceItem(C*, C*)`	Replaces the instance of one item with another one
`bool contains(C*)`	Determines whether or not an item is contained within the list
`void clearList()`	Removes all the items from the list
`void printList()`	Prints the members of the list

3.5.2 `UserDistributions`

`UserDistributions` is a utility class provided by the Möbius discrete event simulator base classes. `UserDistributions` objects provide a set of probability distribution functions and a random number stream. All Möbius models are created with a single `UserDistributions` object (irrespective of what solution method is used to solve the model). There are several places in the base classes where a random number stream is necessary. For instance, the default implementation of `SelectAction` for groups uses a random number to select which one of its members will fire in an enabling state in which there are two or more members in the highest-ranked group. All actions have access to the `UserDistributions` object (`TheDistribution`), since it is declared to be an `extern` data member in `BaseActionClass`. The `UserDistributions` class is important to actions because it greatly facilitates the implementation of the `SampleDistribution` method. Most actions implement the `SampleDistribution` method by calling one of the standard distribution functions defined by `UserDistributions`'s parent class, `Distribution`, with the required distribution parameters. This method take a number from the random number stream and provide a sample point from the specific distribution.

Because it would have been unrealistic to try to implement every distribution function in the `Distribution` class, we defined the `UserDistributions` class with the idea that users may want to implement specific distribution functions that are not already parts of the `Distribution` class. Adding new methods to the `UserDistributions` class would not impact the functionality of any old code, nor would it disturb the base class implementation of the `Distribution` class.

3.6 Additional Functions

In order to make implementation of the `SetState` and `CurrentState` easier, the abstract functional interface includes some utility methods for reading and writing data across word boundaries. The ability to read and write across word boundaries is an important feature that allows an implementation to minimize the amount of memory needed to store model state. These methods are implemented as template functions so that they can be used with any C data type. For completeness, these methods are specified in Table 3.16.

Table 3.16 A list of template functions for reading and writing across word boundaries

Method Name	*Description*
`void readMemory<C>(void* Source, C* Destination)`	This method reads a variable of type C and writes its value to the specified memory location
`void readArray<C>(C* Source, void* Destination, int Size)`	This method reads an array of type C and copies its value to the destination address
`void writeMemory<C>(C Value, void* Location)`	This method writes the value starting from the specified memory address
`void writeArray<C>(C* Value, void Location, int Size)`	This method writes an array of values to the specified memory location

CHAPTER 4

REALIZATION OF AN ATOMIC MODEL FORMALISM THROUGH THE ABSTRACT FUNCTIONAL INTERFACE

This chapter examines how a specific atomic model formalism, namely stochastic activity networks (SANs) [28, 29, 30, 9], has been implemented using the Möbius abstract functional interface. Specifically, we will see how high-level SAN primitives can be expressed using the C++ base class implementation of the abstract functional interface.

4.1 Introduction to Stochastic Activity Networks Theory

SANs are an extension to Petri nets [31] that allow a modeler to build dependability and performance models. SANs are used to define a random process that describes the behavior of a system. Four SAN primitives are used to specify a model in the SAN formalism. Each of these primitives has a graphical representation, which is useful for specifying a model in a concise and intuitive manner. The four SAN primitives are places, activities, input gates, and output gates.

Each *place* holds a portion of the model state as a natural number. The value of a place is called the place's *marking*. The model state is the ordered set of the place markings.

Activities are SAN primitives that represent actions that take some specified amount of time to complete. Activities come in two varieties: *timed* and *instantaneous*. Instantaneous activities take no time to complete, whereas the time to completion for timed activities is expressed as

a random variable. This random variable may be a function of the model state. Activities can also be specified with *cases,* which represent possible outcomes of an activity's completion. Each case is assigned a probability, which can be a function of the model state. Upon activity completion, an activity case is selected based upon the case probabilities and the associated case-specific state change is performed.

Timed activities also have *activation predicates* and *reactivation predicates*. A reactivated activity aborts immediately, and its time-to-completion is governed by a new random variable that is a function of the current state. An activity reactivates if the activation predicate was true when the activity was "activated," the reactivation function is true in a new model state, and the activity remained enabled from the time it is activated to the time the reactivation predicate becomes true. An activity is *activated* if the actitivity becomes enabled or the activity fires and remains enabled in the new state.

SAN *input gates* specify additional enabling conditions for activities. Input gates contain two parts: an "input gate predicate" and an "input gate function." If an input gate is associated with an activity, then the activity is only enabled if all of its associated input gate predicates are true. *Input gate predicates* are Boolean expressions that can be functions of model state. The *input gate function* specifies a change of model state that is executed when its associated activity completes.

Output gates are SAN primitives used to specify additional state functions associated with an activity's completion. These state functions are specified in the *output gate function*. Output gates can also be associated with individual activity cases.

Places and activities may be connected through directed arcs. An arc going from a place to an activity represents an implicit enabling condition. This implicit enabling condition requires that the marking of the attached place be greater than one. A directed arc from an activity (or activity case) to a place specifies an implicit state change upon activity completion. This implicit state change results in increasing the marking of the place by one [32].

4.2 Stochastic Activity Networks Implementation

In order to implement stochastic activity networks in Möbius, we first need to further refine the definition of SANs to make them easy to implement. We will look at each SAN primitive in greater detail and specify additional rules for implemenation.

Our implementation of SAN places will represent their markings as signed `short` variables. This means that the set of valid place markings is limited by the range of a `short` type C variable on the machine used to solve the SAN model. The value of the SAN place can be accessed through the place's marking function. The syntax of this marking function depends upon the modeling tool.

The SAN activity implementation makes use of standard distribution functions as well as the C programming language. For timed activities, the activity's firing time distribution is specified in terms of a probability distribution function whose parameters can be a function of the model state. Thus, in every model state, the activity's firing time distribution parameters are evaluated, returning a random variable used to describe the activity's firing time characteristics. Distribution parameters, case probabilities, activation predicates, and reactivation predicates are all specified using C syntax. Distribution parameters and case probabilities should be specified such that `double` type values are returned, whereas the activation and reactivation predicates should return Boolean values.

Input gate predicates and functions are also specified using C syntax. The predicate is a function of the model state and must result in a Boolean value. The input gate predicate is a C function that specifies a change in the model state. The input gate predicate changes the model state by changing the values of place markings using the Mark function.

Like input gates, SAN output gates also allow sophisticated state changes by means of C functions. However, output gates do not have predicates, and will always execute as a result of the completion of associated activities (or activity cases).

4.3 Realization of SANs Through the Abstract Functional Interface

In order to construct SAN models in the Möbius tool, we first defined a set of formalism base classes that implement the abstract functional interface for stochastic activity networks. By implementing SANs using the abstract functional interface, we hope to show that complex models can be specified and solved using a generic functional interface. The following sections describe how each SAN primitive has been incorporated in our Möbius implementation of SANs.

4.3.1 Places as state variables

Implementing SAN state variables in the Möbius framework was rather easy, since each SAN only has one state variable type: places. The implementation is further simplified by the fact that each place only has a single method to access and change state: `Mark()`. Using the terminology defined in Section 2.4.1, we can say that the `Mark` function is a place's primary value function, and that a place's type is therefore `short`. Furthermore, we would like to implement places in such a way as to allow places to be shared through equivalence relationships with other state variables of type `short`.

Taking all these concerns into account, we derived a class `Place` from the sharable state variable class `SharableSV`. By doing so, we immediately inherited all the equivalence sharing capability encapsulated in the base classes as well as in the default implementation of `SetState` and `CurrentState`. The only additional method needed to implement SAN places as Möbius sharable state variables is the `Mark` method. Historically, SANs have been specified using the syntax `MARK(A)` to refer to the marking of a place named A [33]. However, in the Möbius implementation of SAN places, we must use a method defined on the Place class to access the state value. This requires a syntax that is slightly different from the one used in *UltraSAN* [32]. The new syntax used to reference the marking of a place called A in a SAN model is `A->Mark()`. We implemented the `Mark` method in such a way as to return a

lvalue. By returning a lvalue the modeler is allowed to manipulate the place marking directly. Table 4.1 contains examples of valid SAN syntax as originally described in [32, 9] and how the equivalent statements implemented in Möbius .

Table 4.1 Syntax differences between *UltraSAN* and Möbius

UltraSAN	*Möbius*
MARK(A)	A->Mark() or A->getMark()
MARK(A)++	A->Mark()++
MARK(A) = 4	A->Mark() = 4 or A->setMark(4)

In addition to the `Mark` function we implemented two other functions: `setMark` and `getMark`. These two methods offer another way of interacting with a place's marking. The `setMark` method takes a single `short` parameter and sets the marking of the SAN place to that value. Conversely, the `getMark` method returns the value of the SAN place as a `short` type return value. Using `setMark` and `getMark` may result in code that is easier to understand.

The main motivation for allowing access to a state variable's state only through method interfaces is that it allows us to build functionally sharable state variables. For instance, if we were to define the value of a SAN place to be a function of another state variable, we would implement the functionally shared Place as a derived class that redefines the Mark method. Using this methodology, we do not have to make any changes to the SAN syntax to get the value of a functionally shared state variable. If a state variable's state was a public data member, the syntax used to access the state value of a functionally shared state variable would need to be changed to be compilable.

4.3.2 Activities as actions

Activities are the only actions allowed in SANs. The `Activity` class is derived from `BaseGroupClass`, since stochastic activity networks do not support action groups as they are specified in the Möbius framework; thus, each timed activity without a case is always in a

group by itself. The `Activity` class does not define any of the action's abstract functional interface, since the activity function definitions are specific to individual activities. Thus the `Activity` class is itself an abstract class. Each SAN model creates a new class derived from `Activity` for each of the activities in a SAN model. These specific activity implementations are defined within the scope of the SAN model to remove the prospect of a name conflict caused by a similarly named activity in another atomic model. If these classes were defined outside the scope of the SAN model, then there would be two definitions of the same named class.

The definition of most action methods for SAN activities can be taken directly from the activity specification. Table 4.2 shows which parts of an activity specification are used to define specific action methods. The realization of some particular action methods require structural information about the SAN (which places are connected to each activity).

Table 4.2 SAN sources for action method realizations

SAN Formalism	*Möbius Methods*
Activity activation predicate	`ReactivationPredicate`
Activity reactivation predicate	`ReactivationFunction`
Input gate predicates, Input places	`Enabled`
Activity distribution function, Distribution parameters	`SampleDistribution`, `ReturnDistributionParameters`
Activity case probability functions	`Weight`
Input gate functions, output gate functions, attached places	`Fire`

A modeler defines some activity functions, such as `ReactivationPredicate`, `ReactivationFunction`, `DistributionParameters`, and `SampleDistribution`, via a graphical user interface. There is a direct mapping between the concept of activation predicate in SANs and the `ReactivationPredicate` method defined on Möbius actions. Therefore the definition of SAN activation predicates defines an activity's `ReactivationPredicate` implementation as a Möbius action. Similarly, there is a direct mapping from SAN reactivation predicate to Möbius reactivation functions; thus, the reactivation

```
\* An example implementation of activity that fires with
   an Exponential distribution with rate 10 *\
MySAN::MyActivity::SampleDistribution(){
 return TheDistribution->Exponentional(10.0);
}

\* An example implementation of activity that fires with
   a Gaussian (Normal) distribution with mean equal to the
   current value of a place named A and with
   variance equal to 3.4 */
MySAN::MyActivity2::SampleDistribution(){
 return TheDistribution->Normal((double)A->Mark(), 3.4);
}
```

Figure 4.1 Implementations of the `SampleDistribution` method for SAN activities

predicate specified for an activity definition is used to define the `ReactivationFunction` on `BaseActionClass`.

4.3.2.1 `ReturnDistributionParameters` and `SampleDistribution`

The definitions of the abstract functional interface methods `ReturnDistribution-Parameters` and `SampleDistribution` come from the definition of the activity's distribution function. Each SAN timed activity is specified with a firing time distribution function. This distribution function can be any one of the supported distribution functions in Möbius (see Table 3.8). For each distribution function, a set of distribution function parameters are also specified by the user. The implementation of the `SampleDistribution` method for SAN activities calls a function provided by the simulator library that samples the specified distribution function. (A single random stream that is initialized during model construction is used in the implementation of all supported distribution functions). The value (of type `double`) returned by the simulation library function is used as the return value for the `SampleDistribution` method. Some example implementations are given in Figure 4.1.

```
\*  This method returns a single parameter --
    the rate of the exponential distribution *\
double* MySAN::MyActivity::ReturnDistributionParameters(){
 TheDistributionParameters[0] = 10.0;
 return TheDistributionParameters;
}

\*  This method returns two parameters -- }
    the mean and the variances of the normal distribution *\
double* MySAN::MyActivity2::ReturnDistributionParameters(){
 TheDistributionParameters[0] = (double) A->Mark();
 TheDistributionParameters[1] = 3.4;
 return TheDistributionParameters;
}
```

Figure 4.2 Sample implementation of the `ReturnDistributionParameters` method for SAN activities

The abstract functional interface method `ReturnDistributionParameters` returns the parameters used to define the firing rate distribution function in a given model state. The SAN activity implementation of `ReturnDistributionParameters` evalulates the activity's distribution parameters for the given state and returns them as an array of type `double`. Some sample implementations are given in Figure 4.2. These examples assume that there is a data member called `TheDistributionParameters` in the `Activity` class that is used to return values. This data structure is dynamically allocated when the activity is constructed with a length that is equal to the number of parameters needed for the firing time distribution function.

4.3.2.2 `Enabled`, `Fire`, and `Weight`

The definitions of activities' `Enabled`, `Fire`, and `Weight` methods depend on which places, input gates, and output gates are attached to the activity.

The `Enabled` function is defined as the Boolean expression that combines all the implicit enabling conditions (each input place must have a marking greater than one) and explicit en-

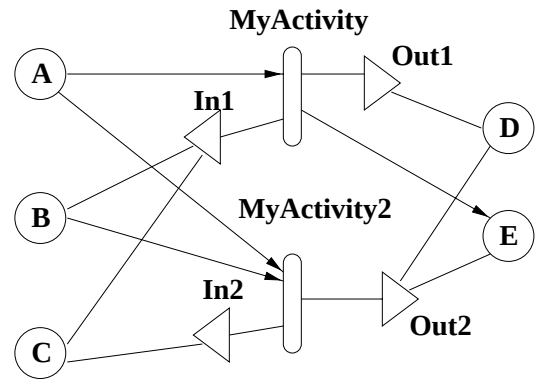
```
\*  This method defines a Boolean expression that
    determines state in which MyActivity is enabled *\
bool MySAN::MyActivity::Enabled(){
 return (A->Mark()>0)&&((B->Mark()>5)&&(C->Mark()<2));
}

\*  This method defines a Boolean expression that
    determines states in which MyActivity2 is enabled *\
bool MySAN::MyActivity2::Enabled(){
 return (A->Mark()>0)&&(B->Mark()>0)&&((C->Mark()>2));
}
```

Figure 4.3 Sample implementation of the `Enabled` method for SAN activities

abling conditions (input gate predicates must be true). All of these Boolean conditions are combined into a single Boolean expression defined in the activity's `Enabled` method. Figure 4.3 shows implementations of this method for the two activites defined in Figure 4.4.

Similarly, the `Fire` function combines the input gate functions, output gate functions, and implicit state changes into a single function. In accordance with SAN execution policy, tokens are removed from the enabling places and the input gate functions are executed first. Then the output gate functions are executed and the implicit token additions take place. Using this methodology, the part of the SAN model specified within input and output gates is absorbed into the definition of the activity in the Möbius model. Examples of how one constructs `Fire` methods for SAN activities are shown in Figure 4.5.

Because activity methods like `Fire` and `Enabled` reference places by name, it is necessary to have `Place` pointers defined within each activity class definition. However, it would be far too wasteful to have a `Place` pointer for every `Place` in the model. Every activity class is defined with only those `Place` pointers used in the definition of the activity. The SAN model editor (the graphical user interface used to specify SAN Models in Möbius) structurally determines what this set of `Place`s is for each activity, and constructs the activity classes accordingly. The SAN model editor also uses the model specification to determine the `Place`s that should be listed in the activities' `EnablingStateVariables` and `AffectedStat-`



In1 Predicate

```
(B->Mark() > 5) && (C->Mark() < 2)
```

In1 Function

```
B->Mark() += 2;
C->Mark() = D->Mark() + 2;
```

In2 Predicate

```
C->Mark() > 2
```

In2 Function

```
;
```

Out1 Function

```
D->Mark() = B->Mark() + 2;
```

Out2 Function

```
if(A->Mark()>0){
   D->Mark() += 3;
}
else{
   E->Mark() += 2;
}
```

Figure 4.4 Two sample SAN activity definitions

```
\*  This method combines the state change
    specified in the input gates and the
    implicit state changes from attached
    input and output places */
BaseActionClass* MySAN::MyActivity::Fire(){

    // Input Place
    A->Mark()--;

    // In1 input gate function
    B->Mark() += 2;
    C->Mark() = DMark->() + 2;

    // Output place
    E->Mark--;

    // Out1 output gate function
    D->Mark() = B->Mark() + 2;

    // return a reference to the activity that fired
    return this;
}

BaseActionClass* MySAN::MyActivity2::Fire(){

    // Input places
    A->Mark()--
    B->Mark()--;

    // In2 input gate function
    ;

    // Out2 output gate function
    if(A->Mark() > 0)
       D->Mark() += 3;
    else
       E->Mark() += 2;
}
```

Figure 4.5 Sample implementations of the `Fire` method for SAN activities

eVariable arrays. The initialization of these data structures is hard-coded into a method called LinkVariables defined on the Activity class.

Using the Place pointers defined in the activity class definition, the activity is free to change the value of the Places by dereferencing the Place pointers and calling the Mark method. However, this way of accessing state is potentially very slow due to the two pointer dereferences (one for the Place and another for the short pointer). To make state accesses faster, one can use "shortcut pointers." Shortcut pointers are short pointers defined in the activity that point directly to a place's marking. If speed is a priority, the SAN model editor can define the activities in such a way that they always use these shortcut pointers to reference state. However, to effectively implement all state accesses with shortcut pointers, all input gates, output gates, case probabilities, and distribution parameters must be parsed to find all state accesses specified by the user. These shortcut pointers are "registered" with the underlying SharableSV class which maintains all the sharing information for the place. If a place is shared with another state variable, SharableSV will change the value of the shortcut pointers to reflect the new location of the state variable state. That is possible because the addresses of all shortcut pointers are "registered" with the underlying SharableSV class.

4.3.3 Multi-case activities as postselection groups

Activities with two or more cases are implemented within the abstract functional interface using postselection groups. Postselection groups are a natural choice for SAN activities with cases, since case selection for activities occurs at firing time. To implement activities with cases, a separate action is created for each activity case. These activities differ only in their Weight and Fire functions (all other functions are the same). The definition of the actions' Weight function corresponds to the case probability definition provided by the modeler during model specification. All of these separate actions are then combined into a single postselection group with a constant rank and weight value of 1. Since all SAN activities are given the same Rank value (1) this makes the probability of selecting a specific case, given that that postselection group was selected to fire, is equal to the activities' Weight value.

4.3.4 SAN models as Möbius models

Lastly, we needed a class to implement the abstract functional interface defined in `BaseModelClass` for stochastic activity networks. This class, `SANModel`, implements the pure virtual methods defined in `BaseModelClass`. Most of the method implementations are trivial in nature (e.g., returning a list of all the actions, groups, and state variables), but some methods are worth mentioning.

The `SetState`, `CurrentState`, and `CompareState` implementations in `SANModel` make use of the fact that SAN model state is always stored as an array of `short`s. For `SetState` and `CurrentState` the `void` pointer passed in as a parameter is casted as a `short` pointer, and the `SANModel` uses `memcpy` to copy the values of individual places to or from memory. `CompareState` similarly casts the parameters as `short` pointers and compares the markings of each place.

To facilitate implementation of many of the `BaseModelClass` methods, the `SANModel` class maintains a master array of activities and places defined in the model.

CHAPTER 5

CONCLUSIONS AND FUTURE RESEARCH

The culmination of the work presented in this thesis and the work done by other members of the Möbius group is a modeling tool based upon an extensible abstract functional interface. As proof, at this date we have already successfully implemented a sophisticated atomic modeling formalism (stochastic activity networks, see Chapter 4), two composition formalisms (Rep-Join and graph-based [21]), a reward variable specification language (performance variables) [34], a discrete-event simulator [22], a Markov process generator [23], and a variety of state-space-based analytical solution techniques. All of these formalisms and solvers are based upon the abstract functional interface presented in this thesis. The multiple solvers attest to the fact that many different solution techniques are possible using only the information provided by the abstract functional interface.

There are several possible additions to the abstract functional interface that would increase the versatility of model specification in the Möbius tool. One such addition would be a clear definition of how to implement structured and unordered state through the abstract functional interface base class `BaseStateVariableClass`. Implementation of both structured and unordered state variables introduces many possibilities for sharing state among dissimilar, structured state variables, and consequently increases the possibility of constructing models expressed in different atomic modeling formalisms. There are also many different issues surrounding the implementation of unordered state variables. For instance, what are the rules for defining an equivalence or functional sharing relationship on an unordered state variable?

Other areas for future research include the development and implementation of new atomic, composed, and connection formalisms. Only after several new formalisms have been integrated

into the tool will we be able to say with confidence that the abstract functional interface truly provides an extensible framework for building heterogeneous models. In the area of composed model formalisms, there are many different ways of composing models that should be researched. For instance, the idea of composing models through a notion of shared actions or events is still very much unexplored. Allowing model compositions in those ways may facilitate the construction of larger models with many complex interactions between different atomic models. Lastly, the Möbius tool does not currently have an implementation of a connection model formalism. Some tools today use this technique of passing results to build heterogeneous models.

All these suggestions for additional research are focused on making the modeling of complex systems a reality. The ability to accurately model complex system wills greatly assist engineers in performance and dependability assessment and validation of critical systems. Much of current and future technology involves overcoming great technological chanllenges and risks by designing enornmous systems far too complicated to be understood by a single person. Fine-grain operational analysis of such systems would be infeasible, time consuming, and expensive. Higher-level analysis, such as stochastic modeling, can model the behavior of complex systems and provide important information with a high degree of accurarcy without a complete, device-level specification. Without increasingly sophisticated modeling tools, increasingly complex designs will becoming harder to assess for important dependability and performance measures.

REFERENCES

[1] D. Daly, D. D. Deavours, J. M. Doyle, A. J. Stillman, and P. G. Webster, "Möbius: An extensible framework for performance and dependability modeling," in *Tool Descriptions of the 8th International Workshop on Petri Nets and Performance Models*, September 1999, pp. 35–39.

[2] W. H. Sanders, "Integrated frameworks for multi-level and multi-formalism modeling," in *Proceedings of the 8th International Workshop on Petri Nets and Performance Models*, September 1999, pp. 2–9.

[3] B. R. Haverkort, "Performability evaluation of fault-tolerant computer systems using DyQNtool+," *International Journal of Reliability, Quality, and Safety Engineering*, vol. 2, no. 4, pp. 383–404, 1995.

[4] R. G. Franks, A. Hubbard, S. Majumdar, J. E. Neilson, D. C. Petriu, J. Rolia, and C. M. Woodside, "A toolset for performance engineering and software design of client-server systems," *Performance Evaluation*, vol. 24, pp. 117–135, November 1995,

[5] M. Veran and D. Potier, "QNAP2: A portable environment for queuing system modeling," in *Modeling Techniques and Tools for Computer Performance Evaluation*, D. Potier, ed., 1985, pp. 25–63.

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

[7] K. C. Chang, R. F. Gordon, P. G. Loewner, and E. A. MacNair, "The research queueing package modeling environment (RESQME)," Tech. Rep. RC-18687, IBM Research Yorktown, NY, February 1993.

[8] S. Christensen and K. H. Mortensen, "Tools on the web," Web site at the Computer Science Department, University of Aarhus (http://www.daimi.au.dk/∼petrinet/tools).

[9] W. H. Sanders, "Construction and Solution of Performability Models Based on Stochastic Activity Networks," PhD dissertation, University of Michigan, 1988.

[10] C. U. Smith, "Integrating new and 'used' modeling tools for performance engineering," in *Computer Performance Evaluation: Modelling Techniques and Tools: Proceedings of the Fifth International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, G. Balbo and G. Serazzi, eds., February 13-15 1992, pp. 153–163.

[11] R. J. Pooley, "The integrated modelling support environment: A new generation of performance modelling tools," in *Computer Performance Evaluation: Modelling Techniques and Tools: Proceedings of the Fifth International Conference on Modelling Techniques and Tools for Computer Performance Evaluation,* G. Balbo and G. Serazzi, eds., February 13-15 1992, pp. 1–15.

[12] R. Fricks, S. Hunter, S. Garg, and K. Trivedi, "IDEA: Integrated design environment for assessment of ATM networks," in *Proceedings of the Second IEEE International Conference on Engineering of Complex Computer Systems*, October 1996, pp. 27–34.

[13] A. P. A. van Moorsel and Y. Huang, "Reusable software components for performability tools and their utilization for web-based configurable tools," in *Computer Performance Evaluation: Lecture Notes in Computer Science*, no. 1469, Berlin: Springer, 1998, pp. 37–50.

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

[15] G. Ciardo and A. S. Miner, "SMART: Simulation and markovian analyzer for reliability and timing," in *Proc. IEEE Int. Computer Performance and Dependability Symposium (IPDS'96)*, September 1996, p. 60.

[16] F. Bause, P. Buchholz, and P. Kemper, "QPN-tool for the specification and analysis of hierarchically combined queueing petri nets," in *Quantitative Evaluation of Computing and Communication Systems: Proceedings of the 8th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (Performance Tools '95) and GI/ITG Conference on Measurement, Modelling and Evaluating Computing and Communication Systems (MMB '95)*, H. Beilner and F. Bause, eds., no. 977 in Lecture Notes in Computer Science, Berlin: Springer, September 1995, pp. 224–238.

[17] D. Daly, D. D. Deavours, J. M. Doyle, P. G. Webster, and W. H. Sanders, "Möbius: An extensible tool for performance and dependability modeling," in *Performance Tools 2000: 11th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, March 2000. To appear.

[18] 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 FastAbstracts presented at the 29th Annual International Symposium on Fault-Tolerant Computing (FTCS-29)*, June 15-18 1999, pp. 15–16.

[19] D. D. Deavours, "Möbius modeling framework." Private communication, 1999.

[20] M. Reiser and S. S. Lavenberg, "Mean-value analysis of closed multichain queueing networks," *Journal of the Association for Computing Machinery*, vol. 27, pp. 313–322, April 1980.

[21] A. Stillman, "Model composition in the Möbius modeling framework," Master's thesis, University of Illinois, 1999.

[22] A. L. Williamson, "Discrete event simulation in the Möbius modeling framework," Master's thesis, University of Illinois, 1998.

[23] J. M. Sowder, "State-space generation techniques in the Möbius modeling framework," Master's thesis, University of Illinois, 1998.

[24] M. A. Marsan, G. Balbo, A. Bobbio, G. Chiola, G. Conte, and A. Cumani, "The effect of execution policies on the semantics and analysis of stochastic Petri nets," *IEEE Transactions on Software Engineering*, vol. 15, pp. 832–846, July 1989.

[25] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*. New York: Addison-Wesley, 1990.

[26] J. M. Doyle, "Implementation of structured and unordered state variable in Möbius," Tech. Rep., University of Illinois, 2000.

[27] A. M. Law and W. D. Kelton, *Simulation Modeling & Analysis*. New York: McGraw-Hill, second ed., 1991.

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

[29] J. F. Meyer and W. H. Sanders, "Specification and construction of performability models," in *Proceedings of the Second International Workshop on Performability Modeling of Computer and Communication Systems*, June 28-30 1993.

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

[31] F. Bause and P. S. Kritzinger, *Stochastic Petri Nets*. Wiesbaden: Vieweg, 1996.

[32] PERFORM Research Group, *UltraSAN User's Manual*, 3.0 ed., Coordinated Science Laboratory, 1994.

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

[34] G. P. K. III, "Design and implementation of an extensible tool for performance and dependability model evaluation," Master's thesis, University of Illinois, 1998.