EXTENDING MESSAGE SEQUENCE CHARTS FOR MULTI-LEVEL AND MULTI-FORMALISM MODELING IN MÖBIUS

By

ZHIHE ZHOU

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

MAY 2002

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of ZHIHE ZHOU find it satisfactory and recommend that it be accepted.

_____
Chair

_____

_____

ACKNOWLEDGEMENT

I would like to thank my advisor, Professor Frederick T. Sheldon, for advice and support on this thesis. I would also like to thank all the members of the SEDS (Software Engineering for Dependable System) Lab group who gave me invaluable assistance.

I would also like to thank Professor Holger Hermanns, who is at University of Twente in Netherlands, for his help in determining this thesis topic.

EXTENDING MESSAGE SEQUENCE CHARTS FOR MULTI-LEVEL AND MULTI-FORMALISM MODELING IN MÖBIUS

Abstract

by Zhihe Zhou, M.S.
Washington State University
May 2002

Chair: Frederick T. Sheldon

Message Sequence Chart (MSC) is a formal language to describe the communication behavior of a system. Möbius is an extensible multi-level multi-formalism modeling tool that facilitates interactions of models from different formalisms. We propose a new version of MSC, Stochastic MSC (SMSC), which is a stochastic extension to the traditional MSC. SMSC is suitable for performability analysis. Mappings from SMSC to Möbius entities are defined so that it can be integrated into the Möbius framework. Together with other formalisms of Möbius, SMSC can be used as a building block for large hybrid models. Users will have additional flexibility in choosing modeling languages in Möbius. Not like other formalisms so far included in Möbius, SMSC has both textual and graphical representations. Modeling with a text editor is the same as writing a traditional program while the graphical representation gives users a direct view of the system.

TABLE OF CONTENTS

ACKNOWLEDGEMENT iii

ABSTRACT iv

TABLE OF CONTENTS v

LIST OF PUBLICATIONS x

LIST OF TABLES xi

LIST OF FIGURES xii

CHAPTERS

1. INTRODUCTION 1

1.1 Problem Definition 1

1.2 Performability Analysis 3

1.3 Modeling Tools 4

1.4 Multi-formalism Tools 5

1.5 Organization of the Thesis 6

2. MÖBIUS FRAMEWORK 8

2.1 Introduction 8

2.2. Möbius Entities and the Abstract Functional Interface 10

2.2.1 State Variables 12

2.2.2 Actions 15

2.2.2.1 The Enabled function and the Fire function 17

2.2.2.2 ReactivationPredicate and ReactivationFunction 18

2.2.2.3 The Rank and Weight Functions 19

2.2.2.4 The SampleDistribution Function 20
2.2.2.5 Action Execution Policy 22
2.2.3 Group 23
2.2.4 The Möbius Models 24
2.3 Hierarchical Model Construction 25
2.3.1 Atomic Model 25
2.3.2 Composed Model 25
2.3.3 Solvable Model 27
2.3.4 Connected Model 29
2.3.5 Study Editor 29
2.4 Solvers 30
2.5 Summary 30
3. 3. MESSAGE SEQUENCE CHART 31
3.1 Introduction 31
3.2 Basic Message Sequence Charts 32
3.2.1 Instances 34
3.2.2 Messages 35
3.2.3 Local Actions 36
3.2.4 Conditions 37
3.3 Event Ordering and Traces 38
3.3.1 Basic rules 39
3.3.2 Coregions 41
3.3.3 General Orderings 42

3.4 Composition of MSCs 43
3.4.1 Vertical Composition 44
3.4.2 Horizontal Composition 44
3.4.3 Alternative Composition 45
3.5 MSC Reference 45
3.6 High-level MSC 46
3.7 Data in MSC 48
3.8 Summary 49
4. STOCHASTIC MSC 51
4.1 Why Stochastic MSC? 51
4.2 Definition of SMSC 53
4.3 Comparing MSC with SMSC 56
4.3.1 Constructs 56
4.3.2 Ordering Rules 57
4.3.3 Traces vs. Processes 60
4.4 The Underlying Stochastic Process 60
4.4.1 SAN 61
4.4.2 SMSC to SAN 62
4.4.3 The Difference Between SMSC and SAN 65
4.5 Summary 66
5. MAPPING SMSC TO MÖBIUS 67
5.1 Motivations and Problem Definition 67
5.2 Identifying State Variables in SMSCs 69

5.2.1 Instance state 70

5.2.2 Conditions 72

5.2.3 Data 72

5.2.4 Special Entities 72

5.2.5 Shareable vs. Non-shareable State Variables 73

5.3 Identifying Actions in SMSCs 74

5.3.1 Local Activities 74

5.3.2 Message Activities 74

5.3.3 Setting Conditions 75

5.4 Expressing SMSC Models in Möbius Framework 76

5.4.1 Deriving SMSC State Variable Classes 76

5.4.2 Deriving SMSC Activity Classes 78

5.4.3 Deriving SMSC Model Class 80

5.4.4 Model Composition 80

5.5 Solving SMSC Models 82

5.6 Summary 82

6. A NETWORK COMMUNICATION EXAMPLE 83

6.1 A Communication System 83

6.2 Model the Stop and Wait Protocol 84

6.3 Modeling the Data Sending and Receiving Processes 87

6.4 A Heterogeneous Model of the Whole System 89

6.5 Summary 90

7. CONCLUSIONS AND FUTURE STUDY 91

BIBLIOGRAPHY 93

APPENDIX 97

A. MÖBIUS GROUPS 97

B. IMPLEMENTATION OF THE MÖBIUS TOOL 101

C. THE MOBIUS SOLVERS 113

LIST OF PUBLICATIONS

- Conference Publications:

1. Zhihe Zhou and Frederick Sheldon. *Integrating the CSP Formalism into the Mobius Framework for Performability Analysis*. in *Proceedings of PMCCS'5*. 2001. Erlangen Germany, Springer-Verlag
2. Frederick Sheldon, HyeYeon Kim, and Zhihe Zhou. *A Case Study: Validation of Guidance Control Software Requirements for Completeness, Consistency and Fault Tolerance*. in *IEEE Proceedings of the Pacific Rim Dependability Conference (PDRC'2001)*. 2001. Seoul Korea.

- Journal Publications:

3. Frederick Sheldon, Gaoyan Xie, Orest Pilskalns, and Zhihe Zhou, *A Review of Some Rigorous Software Design and Analysis Tools*. Software Focus Journal, 2002. 2(4): p. 140-149.

- Papers in Preparation for Submission:

4. Frederick Sheldon and Zhihe Zhou. *"SMSC: A Modeling Language for Performability Analysis*", the 17th IEEE International Conference on Automated Software Engineering, September 23-27, 2002 Edinburgh, U.K.

LIST OF TABLES

Table 1 Rules for constructing all state variable types 12

Table 2 Basic Types and Their Permissible Values 13

Table 3 An example of a state variable 14

Table 4 Functions defined on actions 17

Table 5 Action attributes 17

Table 6 Possible outcomes of interrupt 19

Table 7 Mapping SMSC constructs to Möbius entities 76

Table 8 Data members defined on SMSCInstanceClass 77

Table 9 Data members defined on SMSCActivityClass 79

Table 10 Data members defined on SMSCModelClass 80

Table 11 Methods defined on BaseStateVariable Class 103

Table 12 Methods defined on BaseActionClass 105

Table 13 Action attributes 106

Table 14 Supported distribution functions 107

Table 15 Performance variable related data members 107

Table 16 Methods defined on BascGroupClass 109

Table 17 methods defined on BaseModelClass 110

Table 18 Data members defined on BaseModelClass 112

LIST OF FIGURES

Figure 1 Graphical representation of an MSC 33

Figure 2 Textual representation of the example 1 34

Figure 3 Incomplete messages 36

Figure 4 The condition symbol 38

Figure 5 Example of a Coregion 42

Figure 6 The general ordering symbol 42

Figure 7 An example of HMSC 46

Figure 8 Textual representation of the HMSC 48

Figure 9 An SMSC example 54

Figure 10 Textual representation of SMSC 55

Figure 11 Gates in a SAN 61

Figure 12 Translating an instance to a SAN 62

Figure 13 Translation of an SMSC to a SAN 63

Figure 14 State variables and actions from an SMSC 75

Figure 15 The 4 scenarios of the Stop and Wait protocol 85

Figure 16 The GetFrame SMSC 86

Figure 17 The model of the Stop and Wait protocol 87

Figure 18 The SAN of the sender 87

Figure 19 The SAN of the receiver 88

Figure 20 Construct the system model 89

Dedication

This thesis is dedicated to my wife Ruiyuan for her support and my son Kevin for bringing me the joyful time in my life.

CHAPTER ONE

1. INTRODUCTION

In the past two decades, much research has been conducted in the area of formal methods. Various formalisms have been studied and the corresponding tools developed [1]. The use of formal methods has evolved as the choice to make software and hardware systems, which are undergoing ever-growing complexity, more dependable and of higher performance. However, except for some costly mission/safety critical systems, formal methods are seldom used. Factors that hamper the use of formal methods include initial cost, lack of expertise, etc. One major problem that system engineers face is how to choose an appropriate tool and formalism from a vast array when they decide to adopt formal method(s). Naturally, good tools will facilitate the popularity of formal methods.

1.1 Problem Definition

Message Sequence Chart (MSC) [2, 3] is a Specification Description Language (SDL) widely used in industry for requirement specification, design specification, as well as test case description. MSC is a formal language with a well-defined syntax and semantics. Systems modeled with MSC are decomposed to a number of independent message passing instances. System behavior is specified by a series of charts indicating interactions between those instances.

Performance evaluation is an important branch of formal analysis of system properties [4, 5]. It regards the quality of service a system can provide. However, not all formalisms are suitable for performance evaluation. For example, Petri Net [6] and Process Algebra

[7] cannot be used for performance evaluation[1] although they are two famous formal languages in system liveness, deadlock free, or other static property analysis. MSC is not for performance evaluation either.

The first problem that we addressed in this research is about how we can make MSC suitable for performance evaluation. Petri Net has been extended to Stochastic Petri Net (SPN)[8], which associates stochastic time information to transitions. This extension of Petri Net can be used to address performance measures, and SPN models are widely used for performance evaluation of a system. Similarly, there is an extension to Process Algebra, Stochastic Process Algebra (SPA)[9], in which events are associated with random time information. SPA is also used for system performance evaluation. Based on the same idea, we have extended MSC to Stochastic MSC (SMSC). The newly created SMSC can be used for performance analysis. Although many research works had been conducted [10, 11] after MSC was proposed, no one has tried to extend it with stochastic properties.

The second problem that we addressed in this research is about how to create a tool for analyzing SMSC. We are not going to create a separate tool for SMSC. Instead, SMSC will be integrated into the Möbius framework[12]. Since Möbius is a well-defined framework for multi-formalism modeling and several formalisms (SAN: Stochastic Activity Network[13], PEPA: Performance Evaluation Process Algebra [14], etc.) had been successfully built in [15, 16], SMSC can be easily integrated into Möbius, which enables SMSC to interact with other formalisms in Möbius. By implementing the interfaces required by Möbius, we do not even need to provide analyzers or solvers to the

[1] Stochastic PNs and PAs do, however, provide such capabilities.

SMSC models. The Möbius provided solvers are applicable to solving SMSC models. The SMSC formalism, together with others available within Möbius, can be used for dependability analysis (i.e., performance, availability and reliability or performability analysis).

1.2 Performability Analysis

Performability was coined to include both performance and dependability [17]. Performance is defined as "quality of service, provided the system is correct." Dependability is "the property of a system which allows reliance to be justifiably placed on the service it delivers." Dependability includes reliability, availability, safety and security. In the past, performance and dependability were evaluated separately. However, problems exist when using separate evaluations because system performance actually depends on all of the aforementioned properties. When failures occur in a system, it usually operates at a degraded performance level. Therefore, performance evaluation without taking into consideration dependability does not capture the whole behavior of the system. On the other hand, dependability analysis tends to be conservative because performance considerations are usually not taken into account. To determine the overall quality of service by relating and quantifying aspects of what a specific system is and does (i.e., how well it performs, or performance) with respect to what the system is required to be and do (i.e., how its functionality is affected by faults, or dependability), performability analysis came into existence.

Performability analysis requires that models of the system be built prior to evaluation. Modeling, the process of building models, is the technique that hides the unimportant details while retains the essence of the important aspects of the system to be evaluated,

also known as abstraction. A real system is usually too complex to be analyzed directly. Most commonly, performability analysis is done before the system is actually built. However, at this stage an abstract model is all that is feasible. The abstract model simplifies the system complexity, and yet embodies the same (i.e., at least to the greatest foreseeable extent) structural and behavioral properties, while providing accurate performability predictions of the real system dynamics.

The types of models we are building are based on the formalisms we are using. Generally, all formalisms have well-defined semantics and/or syntax rules. Models from different formalisms have different appearances. For example, a SAN model will be quit different from a PEPA model. A SAN model is a graph, in which circles represent *places*, bars represent *activities*, triangles represent *input gates* or *output gates*, and arcs are used to connect those components. While, in contrast, a PEPA model only consists of a number of lines of texts and symbols that describe the modeled system. No graphical component is included. Although those models could represent the same system, their appearance is usually very different.

1.3 Modeling Tools

Software tools are required to create models and analyze the models for certain system measures. For each formalism, there is one or more software tool(s) available. These tools not only enable users to create models based on the formalisms, but also provide methods of analyzing the models. Some of them even provide a report generator, which can automatically create well-formatted reports.

The Petri Nets formalism has been studied for many years. According to the Petri Nets World website, there are roughly 100 tools registered[18]. These tools deal with various

types of Petri Nets, including timed PN, colored PN, stochastic PN, etc., and can run on any platforms, including Unix, Mac OS, Linux, Windows, DOS, etc.

Queuing networks is another formalism that is often used for performance analysis. Tools based on queuing networks include DyQN-Tool [19], LQNS [20], QNAP2 [21], RESQ [22], and RESQME [23]. For the PEPA formalism, a software tool PEPA Workbench was developed to solve the PEPA models for performance measures [24]. UltraSAN is a tool for specifying and solving SAN modles [25].

1.4 Multi-formalism Tools

In addition to software tools dealing with a single formalism, there are tools that can be used to specify and solve models from more than one formalism. These tools are referred as multi-formalism tools.

Multi-formalism tools can be classified into two categories: software environment that incorporated multiple tools, and integrated multi-formalism modeling tools. Tools in the first category include IMSE (Integrated Modeling Support Environment) [26], IDEAS (Integrated Design Environment for ASsessment of computer systems and communication networks) [27], and Freud [28]. The approach to build such tools is to provide a common user interface with which users can switch from one tool to another.

Tools in the second category aim to build large heterogeneous models by supporting multiple formalisms and solution techniques. One way to implement such a tool is to translate models from different formalism into a single universal modeling language. This is exactly the method adopted by DEDS (Discrete Event Dynamic System) [29]. The second approach is to connect different models by exchanging results. Tools that took

this approach include SHARPE [30, 31] and SMART [32]. The Möbius tool uses a different approach in which a framework is defined and models from different formalisms can share states and results.

1.5 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 describes the Möbius modeling tool in details. The idea for multi-formalism modeling is first introduced. Then, the Möbius entities and their implementations in the Möbius tool are provided. These entities form the base on which the whole idea of the Möbius framework is built. Finally, the method of building and solving models using the Möbius tool is described.

Chapter 3 is about the Message Sequence Chart formalism. A message sequence chart contains a number of MSC components: instances, messages, local actions, etc. These components act as the building blocks of MSCs. A basic MSC describes a simple scenario of system behavior. Several MSCs can be composed together to describe a more complex scenario. The full behavior of a system may be described by a High-level MSC, which contains all the MSCs defined on the system.

Chapter 4 describes our extension to the MSC formalism. Events defined on a Message Sequence Chart are associated with random times, which denote the time needed to finish the events. The extended MSC is called Stochastic Message Sequence Chart, or SMSC.

In chapter 5, we provide a method of integrating the SMSC formalism into the Möbius framework. The components of SMSC are analyzed in order to derive state variables and

actions, which are the Möbius entities necessary to implement a formalism within the Möbius framework.

Chapter 6 discusses an example of a network communication protocol. This example is used to demonstrate how SMSC models can be joined with models from other formalism. In the example, the stop-and-wait communication protocol is modeled as SMSC, while processes sending or receiving data through the stop-and-wait protocol are modeled as SANs.

Chapter 7 concludes this thesis and provides future research directions regarding SMSC and the Möbius tool.

CHAPTER TWO

2. MÖBIUS FRAMEWORK

Software tools for performance, dependability, and performability evaluation of complex computer systems and networks have been widely used, and have contributed significantly to the conceptual integrity and dependability of such systems. The capabilities of such modeling tools have increased greatly over the last two decades. However, this increase is counteracted by the growth in both the complexity of systems to be analyzed and users' expectations of the tools. Modern systems tend to be complex combinations of computing hardware, networks, operating systems, and application software. Therefore, it is difficult, if not impossible, to characterize the performance and/or dependability of such systems using a single modeling formalism or single model solution technique. These challenges call for the development of performance/dependability modeling frameworks and software tools that can predict the performance of such systems.

2.1 Introduction

The Möbius framework is defined, and the Möbius tool developed, by the Performability Engineering Research Group (PERFORM) in the Center for Reliable and High-Performance Computing at the University of Illinois. The Möbius framework provides a method by which multiple, heterogeneous models can be composed together, each representing a different software or hardware module, component, or view of the system. The composition techniques developed permit models to interact with one another by sharing state, events, or results, and are scalable, in the sense that the solution

of an entire model is possible at a cost lower than for an equivalent unstructured model. This framework also supports multiple modeling languages, as well as methods to combine models at different levels of resolution. Furthermore, the Möbius framework supports multiple model solution methods, including both simulation and analysis, and permits the solution of complete models of complex computing and communication systems, and the applications executing on such systems. Finally, the Möbius framework is extensible, in the sense that it is possible to add new modeling formalisms, composition and connection methods, and model solution techniques to the software environment that implements the Möbius framework without changing existing tool components.

The motivation for building the Möbius tool is the fact that no formalism has shown itself to be the best for building and solving models across many different application domains. Similarly, no single solution method is appropriate for solving all models. Furthermore, new techniques in model specification and solution are often hindered by the necessity of building a complete tool every time a novel concept is realized. Hence, the Möbius framework, in which new modeling formalisms and model solution methods can be easily integrated, is defined. In this context, a *modeling framework* is a formal, mathematical specification of model construction and execution. The key problem in implementing the framework is to define an *Abstract Functional Interface (AFI)*, which is realized as a set of functions that facilitates inter-model communication as well as communication between models and solvers. The abstract functional interface also allows the modeler to specify different parts of the model in different formalisms.

The Möbius framework provides a very general way to specify a model in a particular formalism. A formalism is defined as a language for expressing a model within the

Möbius framework, often using a subset of options available within the framework. A model is defined as a collection of some basic Möbius entities, including state variables, actions, reward variables, and groups expressed in some formalism. State variables hold the state information of the model. They could be as simple as integers, or complex data structures. Actions change the state of the model by assigning new values to state variables. They may have a general delay distribution and a general state-change function, and may operate by any one of several execution policies. Reward variables are used to measure something of interest about the model. A group is a collection of actions that coordinate behavior in some specific way. These basic entities are the building blocks of any Möbius model.

Models are classified into certain types. The most basic one is the atomic model. An atomic model is a self-contained model (but not necessary complete) that is expressed 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 of large models. Two or more atomic models (not necessary from the same formalism) can be structurally joined together to form a large model, which is called composed model. It is allowed that a composed model be a component of another composed model. Models can also be loosely connected by sharing solutions. In this case, the model is called connected model.

2.2. Möbius Entities and the Abstract Functional Interface

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

variables are excluded as a basic entity because they are not used for building models but for specifying a way to measure something of interest.

The Möbius tool addresses the issue of allowing multiple formalisms and solution methods within a single tool by requiring models to exchange information with other models and solvers through the abstract functional interface (AFI). The AFI is a set of methods defined on a set of base classes that all formalisms must implement in order to work within the Möbius tool. The AFI is the mechanism by which heterogeneous modeling is possible within the Möbius tool. This interface defines a set of operations necessary for implementing any new formalisms or solvers within the Möbius tool.

In addition to making the tool extensible, the AFI has the added benefit of providing a means for data encapsulation. In short, this means that formalism implementors are free to implement the AFI in the most efficient manner using whatever data structures and algorithms they deem appropriate. Therefore, one important benefit of the AFI and data encapsulation is that formalism implementors decide how they want to store and change model state in their formalisms.

The AFI 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 interact with 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.

The rest of this chapter describes how the Möbius modeling tool is built using the ideas defined in the Möbius framework. The AFI design represents a plan for implementing some of the ideas in the Möbius framework in a software framework. Now, let's see how those basic entities are defined and implemented within the AFI.

2.2.1 State Variables

State variables are used for storing system state. In general, two functions are defined on a state variable: type and value. The function type maps the state variable to a set of possible values that the state variable can take. Theoretically, all possible types can be defined as in Table 1, in which T is the set of all possible types, Z refers to the set of all integers, R refers to the set of all real numbers, and S is a reference to a state variable. The function value returns an element from the state variable's value domain, which is defined by its type, and this element is the value that the state variable currently holds.

Table 1 Rules for constructing all state variable types

1. $Z \in T$	The set of integers
2. $R \in T$	The set of real numbers
3. $S \in T$	Reference to a state variable
4. if $t \in T$, then $2^t \in T$	Any subset of valid types is still a valid type.
5. if $t_1, t_2, ..., t_n$, then $t_1 \times t_2 \times ... \times t_n \in T$	The Cartesian product of valid types is a valid type.

In Table 1, rule1 and 2 define integer and real number as valid types of state variables. Rule 3 says that the reference of state variables is a valid type. This rule is important because it enables a state variable to take a value that refers to another state variable.

Hence, two state variables can take the same value and acts as one state variable. Rule 4 tells us that any subset of a valid type, which is actually a set, is also a valid type. Rule 5 allows the construct of structured state variable types, which contain structured data.

Theoretically, a state variable can take infinite number of values if its type is integer or real. But, when implementing the Möbius tool based on the Möbius framework, the number of possible values must be finite so that the tool can be implemented on our common software platforms that have limited memories. Therefore, the types of state variable are redefined as basic types and derived types. The basic types consist of BOOL, char, int, float, double and short. The permissible values are defined in Table 2.

Table 2 Basic Types and Their Permissible Values

State Variable Types	Permissible values
BOOL	0 (false) and 1 (true)
Char	-128 to 127
Int	-2147483648 to 2147483647
Float	32 bit
double	64 bit
Short	-32768 to 32767

The derived types can be constructed using the rule 3, 4 and 5 defined in the Table 1. The structured state variables allow one to create complex representations of the model state. It also enables the Möbius tool to accept formalisms with rich notation of state.

A state variable is defined by its type and value. In the abstract functional interface, the type is a fixed attribute to a state variable and cannot dynamically change. Once a state variable is defined and instantiated, its type is fixed and defines a set of values that this state variable can hold. The value of a state variable can and will change over time.

The Möbius framework allows a state variable to have one or more value functions. In general, a value function returns the value of a state variable in a particular model state. But a function is just a mapping from one domain to another domain. The returned values may not be the same for different functions even though the state variable holds the same value.

These value functions are further classified as primary value function and secondary value functions. The primary value function defined on a state variable is used to define state equivalence for a state variable. It always returns a value in the range defined by the state variable's type. The secondary value functions defined on a state variable are used to simplify model specification or create functional sharing among two or more models. The secondary functions do not necessary return a value in the range defined by the type of the state variable. But the value returned must be a value in the range defined by the set of all valid types.

The concept of primary and secondary value functions can be illustrated by an example. Suppose a state variable is defined to represent the state of a computer. The type of this state variable is a subset of integers, for example, {0,1,2}. Table 3 shows the state values and their corresponding computer states.

Table 3 An example of a state variable

State Value	Computer State
0	IDLE
1	WORKING
2	DEAD

The primary value function is an identity function that always returns the current sate value, which is 0, 1, or 2. A secondary value function, for example, ***alive*** may be defined on this state variable. The function ***alive*** returns 1 if the state value is 0 or 1, and otherwise returns 0. The secondary value function is a different interpretation of the state value for state variables.

The relationship between primary value function and secondary functions is quite simple. In any two states of a model, if the primary value function returns the same value, each secondary value function must return same value. The value returned by one secondary value function is not necessarily the same value returned by other secondary value functions. This condition ensures that one only needs to test the primary value function when determining whether two state variables are equivalent. Therefore, it is not necessary to test secondary value function.

2.2.2 Actions

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. Möbius actions are 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. 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. The term *fire* means a specific change of a model's state variables defined by the action.

The exact meanings of the action functions are defined in [12], 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 **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.

Table 4 summarizes the action functions for each action in the AFI. In addition to functions, each action has a set of attributes that further define its operation. These attributes are listed and described in Table 5.

Table 4 Functions defined on actions

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

Table 5 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 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

2.2.2.1 The Enabled function and the Fire function

The **Enabled** function determines whether an action can fire. The **Enabled** function is a Boolean expression that evaluates the state variables values and returns either true or false. Based on the **Enabled** function, two new concepts are defined: *Enabling states* and the *set of enabled actions.* An action's *Enabling states* are a set of model states in which the action can change the state of the model, i.e., the **Enabled** predicate is true on these model states. The *set of enabled actions* for a given model state is a set of actions whose

Enabled predicate is true in the model state. The Enabled function allows us to specify conditions under which a specific state change can occur.

If an action is enabled, it may fire by executing the **Fire** function according to the action's execution policy (see section 2.2.2.5). The firing of an action may change certain model state variables' value. The set of all state variables affected by the action's firing is called the action's *affected state variables*. 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 sequences represents the set of all trajectories through the model's state space.

2.2.2.2 ReactivationPredicate and ReactivationFunction

These two functions are used to implement the action interrupt concept of the Möbius framework. The implemented concept is a reduced version of what is defined in the Möbius framework. The interrupt state is defined as a subset of integers in the Möbius framework so that more complicated interrupt policies can be specified. However, in implementing the Möbius tool, this reactivation state was redefined as a Boolean for easier implementation and better efficiency.

The interrupt state of an action is defined by the action's **ReactivationPredicate** function. The **ReactivationPredicate** is a mapping from model states to {true, false}. A "true" value means the action is "interruptible" or "restartable" and might be restated depending on its **ReactivationFunction**. The implementation in the abstract functional interface thus requires that the action function **ReactivationFunction** be evaluated only if the action is restartable.

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.

The **ReactivationFunction** is 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. The action taken upon interrupt depends upon the action's execution policy type, which is summarized in Table 6.

Table 6 Possible outcomes of interrupt

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

2.2.2.3 The Rank and Weight Functions

The Rank and Weight functions are defined on an action for the purpose of ordering the simultaneously enabled actions that are scheduled to fire. When two or more actions are enabled, the firing order is first determined by their rank values. If some actions have the same rank value, then their Weight is used to further decide their firing order.

Rank enables the modeler to implement a priority-based ordering algorithm for scheduling the firing of enabled actions. An action's rank value can also be used to define a priority-based preselection algorithm for action groups. A priority-based selection policy implemented over a set of actions can replace a race-based execution policy.

Each action has a state-specific integer rank value that represents its priority level for the enabling state. Higher numerical values imply higher priority, with 1 being the lowest priority an action can have. 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. Greater weight values imply that an action is more likely to fire. When used in action groups, an action's weight is used to calculate the probability of selecting a representative member from a set of simultaneously enabled, equally ranked members.

2.2.2.4 The SampleDistribution Function

SampleDistribution is a function defined on an action that returns the time-to-completion. The time-to-completion is a period of time in which the action finishes its task. This period of time starts when the action becomes enabled and ends at the time the action fires. Usually, the time-to-completion is a random variable with a certain distribution function. **SampleDistribution** is defined to facilitate model simulation. When using simulation to solve a model, especially the stochastic model, the solver randomly samples from the action's time-to-completion distribution and determines the time to finish the action whenever the action is enabled.

The time-to-completion of an action, which is an important concept that is used in the SMSC extension, may have two different meanings. It may be viewed as the time it takes an action to complete the task it starts working on when the action becomes enabled. Alternatively, time-to-completion may be viewed as a scheduled event that will happen after that period of time. It may not correspond to any task. The first view is often referred to as work-centric notion, and the second as event-centric notion.

For the event-centric notion, the action represents a part of the system that does not have a clear notion of work. When the action is enabled, it merely schedules a state-changing event for some time in the future. The event may not correspond to any "underlying process" that has a concept of partial completion. It may simply represent an explicit state change at a specific time given that the action is not disabled before that specific time point. There would be no need to remember how close it came to the scheduled execution time. An example of an event-centric time-to-completion is the time specified by a timer for retransmission of a data packet in some communication protocols if the sender receives no acknowledgement.

The work-centric view of the time-to-completion regards an action as performing work on a task starting from the time it becomes enabled, and ending when the specific period of time has passed. The task is completed at the end of that period of time. Completion of the task signals that the model is ready to change state. Since the action is performing work on some task, it may be a good idea if we can remember how much work has been done in the case that the action becomes disabled before the completion of the task. The amount of work done is saved in the action's attribute – **FractionComplete**. The amount of work done is quantified by the fraction of time the action was enabled compared to the

sampled time-to-completion. Thus, the value of **FractionComplete** is always between 0 and 1. When a disabled action becomes enabled, the amount of work done must be taken into account in determining the new time-to-completion.

2.2.2.5 Action Execution Policy

Each action also has a specific execution policy associated with it. The execution policy is a set of rules that govern how state changes occur as a function of time. Execution policy is particularly important in stochastic modeling because the actions' time-to-completion is usually described by a continuous time random variable. This means that each of the enabled actions can fire at one of many possible discrete time-points. One important problem regarding actions is what will happen if an action is enabled but does not fire. The execution policy is defined to resolve this problem.

The AFI supports three race-based execution policies for actions. Racing is the phenomenon that two or more actions are all enabled in the same model state and any one can fire first. The next state change is determined by the action with minimum completion time. These execution policies are Race-Resample, Race-Enabled, and Race-Age, respectively.

Race-Resample is the simplest of all three race-based execution polices. An enabled Race-Resample action will do one of two things at the next state change. 1) It completes the action, executes its **Fire** function, and causes the state change. 2) It has not completed the action when a state change occurs, and it loses all the work it has done from when it becomes enabled to the to the point of this state change. If it is still enabled in the new

state, it acts as if it just became enabled, and the new time-to-completion is only a function of the new model state.

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 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's state.

Race-Age is the most complex one of the three execution policies. Actions with Race-Age execution policy can remember the amount of work done during the enabling period. When a Race-Age action becomes disabled because another action has completed and the model state has changed, it saves the amount of work done in its attribute - **FractionComplete**. When the action is re-enabled in the future, its FractionComplete is used to change the time-to-completion to reflect the fact that work has been accomplished.

2.2.3 Group

Groups are another entity of the Möbius framework. Since groups are not used when we map SMSC to the Möbius framework, we do not provide detailed description here. Refer to Appendix A for more information about groups.

2.2.4 The Möbius Models

Given these basic entities, a model in the Möbius framework is a collection of state variables, actions, groups, and all functions defined on them. A model defines the behavior or a part of the behavior of a system. The system behavior is specified in terms of system state and the change of the system state as a function of the current system state and time. A model is essentially a container for state variables, actions and action groups. Models also have states and types. The type of a model is formalism specific and is defined by the formalism implementing the model.

The model state consists of state variables state, actions state, and group state. The state of the state variable is defined by the value that the state variable currently holds. The action state includes the enabling status of the action in the previous model state, reactivation status, and **FractionComplete**. The group state contains information about what members did in past states. More specifically, the group state stores information about which members were selected in the previous states.

Models can be combined to form larger models. A large model can be recursively decomposed to obtain the final set of state variables, actions and groups. The state of these entities is used to determine the state of the large model.

Models define functions that are used to perform certain operations on the state variables, action, and groups contained in the models. 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. Three functions, **ListActions**, **ListGroups**, and **ListStateVaribles**, are used to return a set of Möbius

entities corresponding to the function names. **SetState** is used to write the model state, usually the initial model state. **CurrentState** is for reading the current model state.

2.3 Hierarchical Model Construction

The Möbius framework is defined not only to support multiple formalisms, but also to enable the construction of large heterogeneous models hierarchically. Models in the Möbius framework are divided into four categories: atomic models, composed models, solvable models, and connected models. SMSC models can be used as atomic models in the Möbius framework. They can also be joined with other models to form composed models.

2.3.1 Atomic Model

An atomic model is a model that is built from a single formalism. Atomic models form the lowest level of the model hierarchy. Every large model must begin with the building of atomic models. Atomic models are building blocks for any other high-level models. An atomic model is often used to model a small part of the system. Although an atomic model does not need to be complete, it must be self-contained.

2.3.2 Composed Model

Two or more atomic models can be jointed together to form a composed model. In addition, a composed model may itself be joined with other atomic or composed models to form a new composed model. A composed model can be decomposed recursively to obtain a number of atomic models. These atomic models are not necessarily from the same formalism.

Models are joined together by sharing certain state variables. The most popular type of sharing relationship between two state variables is the equivalence sharing. Two state variables are said to have an equivalence sharing relationship if they have the same type and always hold the same value. If the value of one state variable changes due to action firing, the value of the other one changes accordingly. As a result, these two state variables still maintain the same value. When the two shared state variables belong to two different models, we say these two models are joined together by sharing the state variables.

The state variables retaining an equivalence sharing relationship require that the same value be assigned to each of them. Since they all have the same value, it is adequate to keep only one copy of the state variable's value in memory, and this copy is shared by all these state variables. This is indeed the method used by the Möbius tool to implement shared state variables. In the class BaseStateVariableClass (Appendix B), there is no data member defined for keeping the state variable's value. It is the requirement of formalism implementors to derive their own state variable classes and define the data member for holding state variable's value. This data member must be declared as a pointer pointing to a variable with the type defined for this state variable. Note that every state variable has a type associated with it. When allocating memory for state variable value pointers, only one of the shared state variables' pointers is assigned memory. All other pointers are set to point to the same memory area. Thus, the change of any shared state variable is reflected to any other shared state variables.

Equivalence sharing requires that the shared state variables have the same type. Atomic models from the same formalism are easily joined through equivalence sharing

because the state variable's types are consistent. There are no unknown state variable types between these models. If two heterogeneous models contain the same type of state variables, these state variables could be used to join the models through equivalence sharing.

Another way of joining heterogeneous models through state variables is functionally sharing state variables. For functionally shared state variables, one state variable's value is define as a function of another state variable's value. There are two types of functional sharing: unidirectional, and bi-directional.

Unidirectional functional sharing is an asymmetrical sharing relationship in which one state variable receives value from a function defined on another state variable. The receiving state variable can only look at its value. It has no method defined to change its value. This state variable is called the read-only state variable. This restriction is removed for bi-directional functional sharing in which the function is defined as one-to-one and onto. Actually, the equivalence sharing relationship is a special case of bi-directional functional sharing in which the function is an identity function.

2.3.3 Solvable Model

The purpose for modeling a system is to obtain certain performance, dependability, and performability measures through analysis of the model. The Möbius tool uses reward models (also known as reward functions) as a way of measuring such properties. After performance variables are defined for the atomic or composed models, these models are called solvable models, meaning they are ready to be solved for certain system measures.

Performance variables are reward variables defined on models. Reward variables can represent any aspect of the system. They are specified in terms of a common, uninterpreted unit of measure, called a unit of reward. The actual meaning of a reward variable is determined by giving "reward" a more specific interpretation. For example, one reward variable could represent the number of jobs in a buffer, and another one the number of working servers in a queue model.

There are two types of reward variables: impulse reward variables and rate reward variables. Impulse reward variables are associated with actions. The impulse reward variable may obtain a unit of reward whenever the action to which it is associated fires. Rate reward variables are associated with the time spent in a model state. They can be used to measure how long the system is in certain state within a given period of time.

Reward variables can be specified to be measured at an instant of time, or accumulated over an interval of time. The first kind of reward variables is also called instant-of-time reward variables, and the latter interval-of-time reward variables. For a rate reward variable, the instant-of-time measure takes the value of the reward variable at that moment. The instant-of-time measure for an impulse reward variable at time t is the value that the reward variable held when the action associated with reward variable completed before time t. As for interval-of-time measures, a rate reward variable accumulates during the time within the period that the model is in the state to which the reward variable is associated, and an impulse reward variable accumulates each time the action fires within the period. If the accumulated value of a reward variable is divided by the length of the time period, the reward variable is called time-averaged interval-of-time reward variable.

Reward variables are actually random variables, which implies they can take different values with different probabilities. Thus, we can speak of the distribution of a reward variable as well as its mean and variance. These properties are of our interest when measuring system properties.

2.3.4 Connected Model

It is not the case that models can always be joined together through sharing state variables. Some heterogeneous models may have quit different state variable types. It could be very difficult to even define functions for functionally sharing state variables. In other words, some models may not have a relationship in terms of state variables. However, being parts of the same system, models do have certain relationships. For instance, the average input data rate for a buffer should be equal to the average output data rate. In other words, these heterogeneous models have certain performability measures in common. The Möbius framework defines the connected model as models that share certain solutions. To distinguish it from the composed model, the connected model contains models that are loosely connected.

2.3.5 Study Editor

The Study Editor is used to assign a value or range of values to global variables defined in the model. Accordingly, it can generate a series of experiments, and each of them corresponds to a certain setting of global variables. The different settings represent the different model parameters. Reward variables of the model can be solved for each experiment. In this way, sensitivity analysis is conducted, which enables the modeler to

know which model parameters most significantly affect the performability measures, and what are the optimal parameter settings needed to achieve the most benefit.

During the specification of atomic, composed, and reward models, global variables are often used to characterize model characteristics. However, the global variables are not given values during the model construction process. Valid global variable types are any C++ basic types, including int, short and double. Only after the global variables are assigned values can a model be solved.

2.4 Solvers

The process of using software tools to analyze a model for the purpose of obtaining certain performability measures from the system under study is called solving the model. The software tool used to calculate (either numerically or analytically) the measure of a model property is called a solver. The Möbius tool provides two classes of solution techniques: discrete event simulation and analytical/numerical technique. Refer to Appendix C for detailed information about how to use these solvers.

2.5 Summary

In this chapter we discussed the Möbius framework and the Möbius tool implemented based on the framework. Up to this point, one should have a general idea about how models are expressed in the Möbius framework, how a formalism is specified within the framework, and the constraints or rules about how heterogeneous models interact with each other as well as the solvers.

CHAPTER THREE

3. MESSAGE SEQUENCE CHART

3.1 Introduction

Message Sequence Charts (MSC) [3] is a language that is used to describe the interaction between a number of independent message-passing instances. According to [3], the main characteristics of the MSC are the following:

- MSC is a scenario language. An MSC describes the order in which communications and other events take place.
- MSC supports complete and incomplete specifications. It has the possibility to describe incomplete behaviors used in early analysis and for documentation purposes.
- MSC is both a graphical and a textual language. The two-dimensional diagrams give an overview of the behavior of communicating instances. The textual form is used for exchange between tools and as a basis for automatic formal analysis.
- MSC is a formal language. The definition of the language is given in natural language as well as in a formal notation.
- MSC is a practical language. MSC can be used throughout the engineering process. Its use ranges from domain analysis and idea generation via the requirements capture at design phase to testing. MSC is used in slightly different ways in the various phases, and it is important that MSC has formal

expressive power as well as an intuitive appearance (i.e., providing behavioral visualization).

- MSC is widely applicable. It is not tailored for one single application domain.
- MSC supports structured design. Simple scenarios described by Basic Message Sequence Charts can be combined to form more complete specifications by means of High-level Message Sequence Charts. MSCs are gathered into an MSC document. A modular design of scenarios is supported by mechanisms for decomposition and reuse. This feature fits perfectly into the Möbius framework for hierarchical model composition.
- MSC is often used in conjunction with other methods and languages. Its formal definition enables formal and automated validation of an MSC with respect to a model described in a different language. MSC can, for example, be used in combination with SDL (i.e., Specification Description Language).

The usual interpretation of a scenario specified in an MSC is that the actual implementation should at least exhibit the behavior expressed in the scenario. Alternative interpretations are also possible. An MSC can, for example, be used to specify disallowed scenarios.

3.2 Basic Message Sequence Charts

Basic Message Sequence Chart (BMSC) is the core of the MSC language. A BMSC describes a simple scenario of a part of the system behavior. The three important primitives for BMSC are instances, messages, and local actions. They are all highly abstract and can be used to represent objects in many different application domains.

An MSC can be expressed graphically or textually. Figure 1 shows an example of a graphical representation of one Basic MSC. The MSC is drawn as a frame containing the instances. The key word **msc** is followed by the name of the MSC and is placed inside the frame near the upper-left corner. Three instances exchange several messages with each other as well as with their environment. The environment is an imagined instance capable of sending and receiving messages. The instance *i1* also performs a local action.

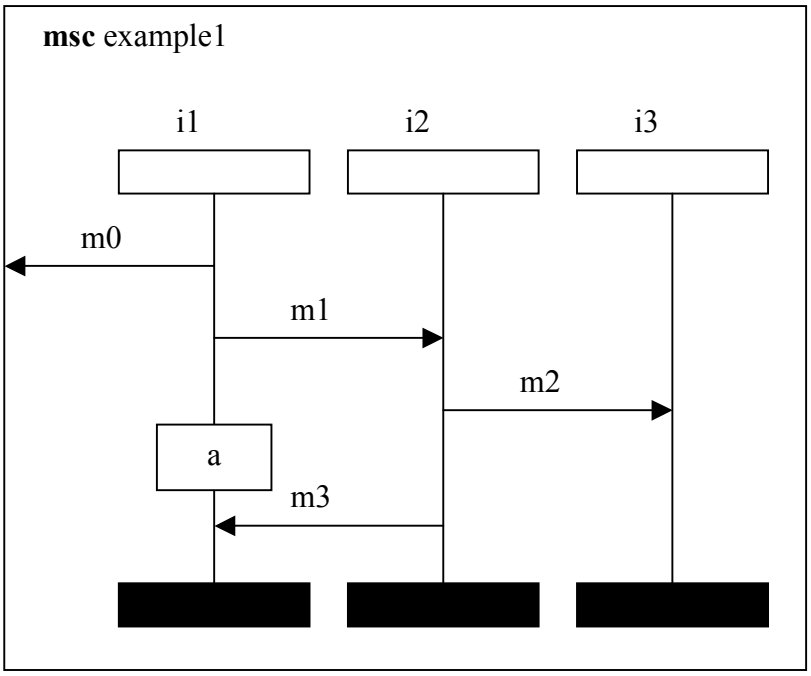


Figure 1 Graphical representation of an MSC

The textual representation can be done in two ways. First, an MSC can be described by giving the behavior of all its instances in isolation. This way of describing an MSC is called instance-oriented. Another way of representing an MSC is the so-called event-oriented description. With the event-oriented descriptions, a list of events is given as they are expected to occur in a trace of the system or as they are encountered while scanning the graphical MSC from top-to-bottom. The instance-oriented description of the same

example is shown in Figure 2 (a). Figure 2 (b) shows the event-oriented description of the MSC. The keyword **msc** denoting the beginning of an MSC is followed by the MSC name. The MSC ends with the keyword **endmsc**.

```
msc example1;
  instance i1;
    out m0 to env;
    out m1 to i2;
    action a;
    in m3 from i2;
  endinstance
  instance i2;
    in m1 from i1;
    out m2 to i3;
    out m3 to i1;
  endinstance;
  instance i3;
    in m2 from i2;
  endinstance;
endmsc;
```

(a)

```
msc example1;
i1: out m0 to env;
i1: out m1 to i2;
i2: in m1 from i1;
i2: out m2 to i3;
i3: in m2 from i2;
i1: action a;
i2: out m3 to i1;
i1: in m3 from i2;
endmsc;
```

(b)

Figure 2 Textual representation of the example 1

3.2.1 Instances

Instances are the primary entities in an MSC. A Message Sequence Chart is composed of interacting instances. In specific systems, an instance may represent a system component, for example, a process or a service. Within the instance body the ordering of events is specified.

Graphically, an instance is drawn as a vertical line starting with the instance head symbol and ending with the instance end symbol. The instance head symbol is a rectangular box, and the instance end symbol is a solid rectangular box. These symbols describe the beginning and ending of the instance within the MSC. They by no means

define the creation and termination of the instance. Each instance has a name associated with it. The name can be placed inside the instance head symbol or above it. In the system level, all instance fragments with the same name constitute the same instance.

The textual representation of an instance begins with the keyword **instance**, and ends with **endinstance**. Between theses key words are ordered events defined for the instance. This representation is mainly used in an instance-oriented description. When using event-oriented descriptions, one only needs to specify the instance name for each event. The instance name is followed by a colon, and then by an event attached to the instance. See Figure 2 (b) for an example.

3.2.2 Messages

Instances in an MSC interact with each other by exchanging messages. The graphical description of a message is an arrow that starts at the sending instance and ends at the receiving instance. A message sent to the environment is represented by an arrow from the sending instance to the surrounding frame. In the case that a message is lost, i.e., the message is sent but never consumed, the arrow ends at a black dot, which denotes a "black hole." Symmetrically, a message can be found, meaning it originates from nowhere. In this case, the arrow starts at an open dot ("white hole"). A lost or found message is called incomplete message because there is either no sending instance or receiving instance associated with the message.

A message exchange involves two events: the event of sending the message and the event of receiving it. Thus, the textural representation of a message consists of two event descriptions. The sending event is described as:

out <message_name> **to** <input_address>

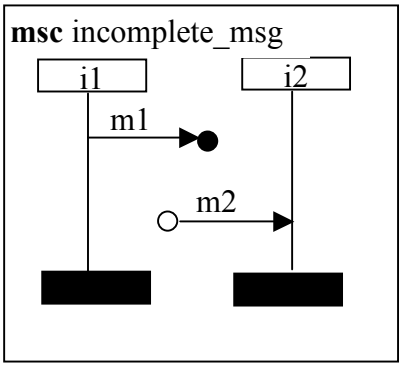
where **out** and **to** are the keywords, <message_name> is the name of the message, and <input_address> is either the name of the instance that consumes the message or the keyword **lost** if this message is an incomplete one.

The event of receiving a message is described as:

in <message_name> **from** <output_address>

where <output_address> is either the name of the instance that sends the message or the keyword **found,** which means that the message has no sending instance associated with it.

An example of incomplete messages is shown in Figure 3. The left side is the graphical representation, and the right side is the corresponding textual description.



msc incomplete_msg;
i1: **out** m1 **to lost**;
i2: **in** m2 **from found;**
endmsc;

Figure 3 Incomplete messages

3.2.3 Local Actions

In addition to message exchange the local action may be specified in MSCs. A local action describes an internal atomic activity of an instance. It contains either informal text

that describes the internal activity, or a formal data statement that defines operations on some data.

Graphically, a local action is denoted by an action symbol on an instance with the action string placed in it. The action symbol is a box placed on the instance axis. The part of the axis covered by the action symbol turns invisible or is removed.

The textual representation is very simple. A local action is textually described by the keyword **action** followed by the action string or data statements. Refer to Figure 1 andFigure 2 for the two types of representation for local actions.

3.2.4 Conditions

Besides instances, messages, and local actions, another important construct for Message Sequence Charts is condition. Conditions can be used to restrict the traces that an MSC can take. There are two types of conditions: setting and guarding conditions. A condition can be a global condition shared by a number of instances or a local condition attached to only one instance.

Setting conditions are used to set or describe the current global system state if the condition is global, or some non-global state if the condition is not global. Guarding conditions restrict the behavior of an MSC by only allowing the execution of events in a certain part of the MSC depending on their values.

The value of a guarding condition is either a state that can be set by a setting condition, or a Boolean expression. When the Boolean expression evaluates to “true” or the system is in the state required by the guarding condition, we say the condition is met.

The events in the scope guarded by the guarding condition can only be executed if the guarding condition is met.

A graphical condition symbol is defined as shown in Figure 4. For a setting condition, the condition name is placed in the condition symbol, meaning set the system to a particular state specified by the condition. For a guarding condition, the keyword **when** is placed in the symbol, and the condition name or a Boolean expression follows the keyword. It means when the system is in certain state or the Boolean expression is true, the events following the guarding condition can be executed.

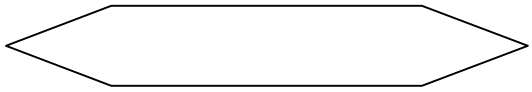


Figure 4 The condition symbol

The textural representation of a setting condition is the keyword **condition** followed by the condition name. A guarding condition has the keyword **when** inserted between them.

3.3 Event Ordering and Traces

A Message Sequence Chart is intended to describe a number of executions of events. These events are either message exchanges among instances or local internal actions. One assumption about the execution of events is that all events are executed instantaneously, i.e., the execution of events consumes no time. Another important assumption is that no two events can be executed at the same time. Under these two assumptions, a message sequence chart in fact describes the possible orders in which the events can be executed.

To order the events, we must define ordering rules for Message Sequence Charts. The order in which events take place must comply with these rules. If we describe an execution of an MSC by a sequence of events in the order that they take place, this sequence forms a trace of the execution. A message sequence chart specifies the behavior of a part of the system through a set of valid traces. A valid trace is a sequence of events where the order of such events does not violate any ordering rules. The complete set of valid traces defines the possible behavior that this part of the system can have. An invalid trace is the one where its event order violates at least one ordering rule.

The MSC language defines basic ordering rules, general ordering constructs, and the concept of coregions in the ordering of all events. The basic ordering rules are further described below.

3.3.1 Basic rules

There are two basic ordering rules. The first rule deals with the ordering of events of the same instance. This rule says that **the events of an instance are executed in the same order as they are given on the vertical axis from top to bottom**. One can say that the time along each instance axis is running from top to bottom. Therefore, the events specified on an instance are totally ordered in time. However, there is no scale of time associated with the instance axis. The instance only specifies the order for the events. It does not specify the elapse of time in between two consecutive events. It is possible that the first event is executed at 10 seconds and that the second event is executed at 30 seconds or any time point that is later than 10 seconds.

The second rule deals with the order imposed by messages. The key idea for defining this rule is that a message must be sent before it can be consumed. Intuitively, this is

obvious since a message cannot be received before it is sent. Therefore, the second rule is defined as **the event of sending a message must happen before the event of receiving the same message**.

In principle, the instances operate independently. No global notion of time is assumed. The only dependencies between the timing of instances come from the restriction of the second rule. In the example shown in Figure 1, this implies that message *m2* is received by *i3* only after is has been sent by *i2*, and, consequently, after the consumption of *m1* by *i2*, while for local action *a* and message *m2* no order is specified. It is therefore possible that action *a* is executed before *m2* is sent, or after *m2* has been received, or even between the sending and receiving of *m2*. Furthermore, because the asynchronous communication assumption, even the order of local action *a* and the consumption of message *m1* is not specified. Local action *a* could be executed before or after the consumption of m1 by *i2*. But it can only be executed after *m1* has been sent by *i1*. We can see that the execution of local actions is only restricted by the ordering of events of the instance it is defined on.

The second rule also implies that a message output is not allowed to depend on its corresponding message input, directly or indirectly via other messages. In this case the Message Sequence Chart would specify that a message be received before its corresponding message is output. Such Message Sequence Charts are called inconsistent.

Before we introduce any other concepts regarding the ordering of event, it is important to note that these two ordering rules imply that events defined on an instance can be totally ordered, while events between different instances may not be totally ordered. If two instances have no message exchange, we have no way to order the events between

them. Therefore, a Message Sequence Chart only imposes a partial ordering on the set of events being contained.

3.3.2 Coregions

So far the events of an instance are totally ordered in time according to the first ordering rule. In reality, it is quite possible that some events can happen in any order, i.e., the ordering of them does not matter in terms of the behavior of the system being specified. The concept of coregion is introduced to enable the specification of unordered events for an instance.

To represent a coregion in a Message Sequence Chart, a new construct is defined. Graphically, a coregion can be drawn as a dashed vertical line that replaces the part of the vertical line representing the instance. Message arrows starting from or ending at the coregion define message events for the coregion. Local actions can also be placed in the coregion. Textually, a coregion is enclosed between two keywords: **current** and **endcurrent**. The keywod **current** begins a coregion, which is ended by **endcurrent**.

Events defined on a coregion can take place in any order. This contradicts the first ordering rule. So the first ordering rule is revised as follows:

- **The events of an instance that are not in a coregion are executed in the same orders as they are given on the vertical axis from top to bottom. Events inside a coregion can be executed in any order, but these events can only happen bwtween the execution of the event specified immediately before the coregion and the execution of the event specified immediately after the coregion if such events exist.**

Figure 5 shows an example of coregeion.

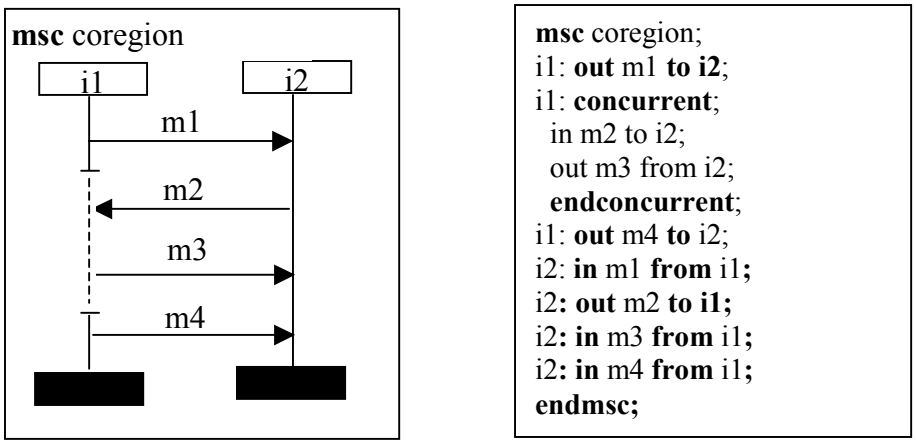


Figure 5 Example of a Coregion

3.3.3 General Orderings

General orderings are introduced to describe the ordering between events when this ordering cannot be derived from the MSC according to the two basic ordering rules. In other words, general orderings are used to specify the ordering of events that have no direct or indirect relations. The two events associated with a general ordering can otherwise happen. In this sense, general orderings provide additional information to a Message Sequence Chart.

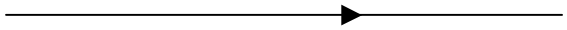


Figure 6 The general ordering symbol

A general ordering is graphically denoted by the general ordering symbol, which is a solid line with an arrowhead in the middle as shown in Figure 6. The symbol distinguishes the general ordering from normal messages where the arrowhead is always placed on one end of the line. Each of the two ends of the general ordering symbol is

attached to an event. The arrowhead in interpreted as pointing to the event that happens later. Therefore a general ordering specifies the ordering of executions of two events in a similar way as a message does. To facilitate a textual grammar for general ordering, keywords **before** and **after** are defined.

Usually, general orderings are used to specify the ordering of two events defined on different instances since the ordering of these events is often not defined. In a rare case that an instance has no communications with other instances, the ordering of events between this instance and those of others can only be specified through general orderings.

General orderings can also be used to describe the ordering of events from the same instance. When a general ordering is applied to two events in the same coregion, it does give additional information. A general ordering will explicitly specify the ordering of the two events in a coregein. General orderings for events outside coregions are allowed. But they are really not necessary since the ordering of these events is already defined.

The use of general ordering must not cause conflicts. If the ordering of two events is defined directly or can be derived indirectly, the general ordering applied to the events should specify the same order. Otherwise, it introduces inconsistency.

3.4 Composition of MSCs

MSCs can be composed together to specify the behavior of large systems. The MSC language defines composition operators that specify the behavior of MSCs when they are put together. The three primary methods of combining MCSs are vertical composition, horizontal composition, and alternative composition.

3.4.1 Vertical Composition

The vertical composition of two MSCs means placing one MSC at the bottom of another one and then connecting the instances they have in common thus obtaining a new MSC.

In the new MSC, the ordering of events should be interpreted as follows:

- For the common instances, events from an instance of the bottom MSC should happen after those from the same instance of the top MSC.
- For the different instances, no new restriction is imposed by this composition. The ordering of the events is the same as if these instances were already in the same MSC.

Therefore, vertical composition does not necessarily mean that all events in the bottom MSC must happen after all events in the top MSC. Only events of the common instances are affected by the vertical composition.

3.4.2 Horizontal Composition

The horizontal composition of two MSCs means placing them next to each other. The behavior of the instances in common is the interleaving of the behaviors of these instances in the separate MSCs. The events of a common instance are ordered in such a way that the order of events imposed by the MSC before composition is preserved. If two MSCs have no common instance, their vertical composition is the same as horizontal composition.

3.4.3 Alternative Composition

The alternative composition of two or more MSCs means a choice has to be made between or among them. The chosen MSC governs the further behavior of the system under study.

In complex systems there are many points of deviating behavior. It is important to be able to indicate at which point alternatives occur. Alternatives are specified by different MSCs. The MSC language offers the alternative composition operator to cover such needs. An important aspect of the alternative mechanism in MSC is that the moment of choice between different scenarios is postponed until that choice can no longer be avoided.

3.5 MSC Reference

In an MSC one can refer to another MSC by means of an MSC reference. An MSC reference refers to another MSC by the name of the referred MSC. Therefore, the name of an MSC must be unique. It is not allowed that two or more MSCs bear the same name.

Graphically an MSC reference is represented by a rounded frame with the referenced MSC name placed in it. The MSC reference symbol is drawn on top of a number of instances. To ensure a clearly defined event order for those common instances, there are two requirements that must be satisfied:

- If an instance of the enclosing MSC is also present in the MSC reference, then the MSC reference symbol must overlap this instance. An instance is present in an MSC reference if at least one of the MSCs that are referenced in the expression has an instance with that name.

- If two MSC reference expressions in the same enclosing MSC share an instance then this instance must be drawn in the enclosing MSC.

Intuitively, an MSC reference can be replaced by the referenced MSC. This process can be done recursively until we have a MSC without any MSC reference in it. The ordering of events in this MSC is the same as that of the MSC with MSC references.

3.6 High-level MSC

The High-level MSC (HMSC) provides another way to combine Message Sequence Charts. The graphical representation of a high-level MSC is a directed graph, where the nodes are formed by MSC references or another high-level MSC and arrows imply an order on the nodes. An example of HMSC is shown in Figure 7.

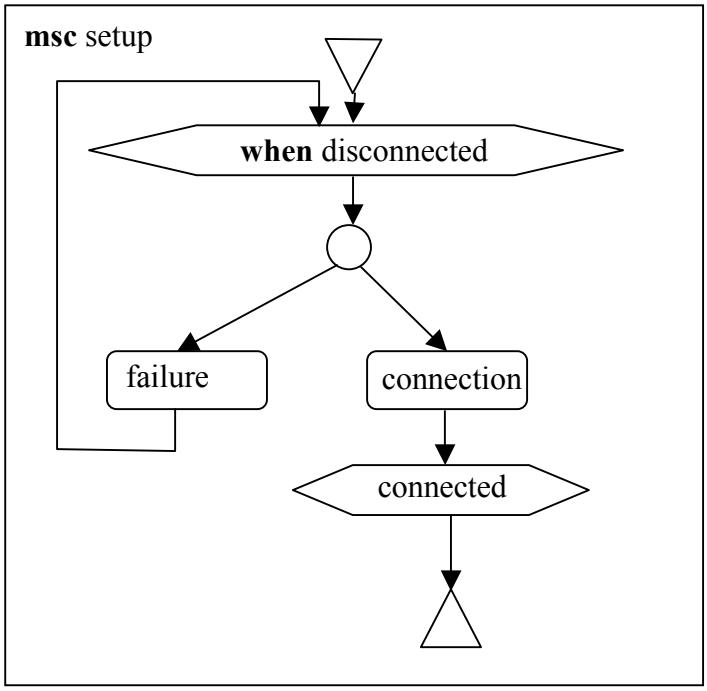


Figure 7 An example of HMSC

MSC references and conditions are valid constructs in an HMSC. MSC references are the primary constructs of an HMSC since the goal of creating HMSC is to organize a number of MSCs. Conditions are used to restrict the behavior of the HMSC. An HMSC is also a MSC in that an HMSC also describes the behavior of some system. Thus, it can be referenced in another HMSC. There are other elements defined for HMSC. Every HMSC must start with a start node, which is graphically represented by an upside-down triangle (▽). Every other node must be reaching from the start node. Each node, except for the end node (△), must have a successor. A node can also be a connection node, which is graphically represented by a circle (○). The meaning of connection nodes is void. They are used to distinguish crossing lines from splitting lines.

It is easy to explain the meaning of an HMSC. If two MSC reference nodes are connected via exactly one arrow, they are vertically composed. The arrow always points to the bottom MSC. If an MSC reference has more than one outgoing arrow, then all the successors of that node are alternatives for the vertical composition with that MSC reference node. Horizontal composition can be specified by a parallel frame. A parallel frame can contain more than one state node. Each state node indicates an operand for the horizontal composition operator.

The HMSC also has textual representations. Figure 8 shows the textual representation of the example shown in Figure 7. We can see that the keyword **seq** denotes the vertical composition, while **alt** means alternative compositions. Each node is defined as a label in the textual representation.

```
msc setup;
expr L1;
  L1: condition when disconnected seq (L2);
  L2: connect seq (L3 alt L4);
  L3: (failure); seq (L1);
  L4: (connection); seq (L5);
  L5: condition connected seq (L6);
  L6: end;
endmsc;
```

Figure 8 Textual representation of the HMSC

3.7 Data in MSC

The MSC language also has the ability to declare and operate on data. Data is incorporated into the MSC language by allowing a number of primitives to define operations on data, such as local actions, messages, and MSC references. There are two types of data: static and dynamic. The value of static data can only be assigned once, but can be read many times. The dynamic data consists of MSC variables that can be assigned and reassigned values.

Data statements can be defined on local actions. These statements include the declaring of variables, assigning values to the declared variables, and undeclaring variables. Note that a variable must be declared before it can be used. A variable can be undeclared, meaning the termination of its existence. After a variable is undeclared, any operation on that variable is illegal. Local actions are used to dynamically declare and undeclared a variable.

Messages can also be associated with data as their parameters. The result of a message exchange is that the instance receiving the message receives the data associated with the message by assigning the value to a variable defined on the instance.

An MSC can have a formal parameter list defined on it. When it is referenced in another MSC, the corresponding MSC reference must define a list of actual parameters. The parameters defined for an MSC are always static. Their values cannot be modified by data statements inside this MSC. Data expressions are allowed to reference the value of a parameter. Like function calls in C, the formal parameters are substituted by the actual parameters of an MSC reference. The values of actual parameters are defined in the MSC that encloses the MSC reference.

The scope of a variable can be global, local MSC, or local instance. Global variables are accessible to any MSC and any instance in an MSC. A variable defined as local MSC is only accessible to instances within the MSC. No other MSC can see the value of the variable. If a variable is defined as local instance, only that instance can access the variable. The smallest scope of a variable is an instance. A variable can also be shared by several instances.

3.8 Summary

In this chapter, the MSC formalism was introduced, which is closely related to our stochastic extension version of MSC - SMSC. SMSC will be defined in the next chapter. A Message Sequence Chart describes the interaction between a number of instances through message passing. An instance can also perform internal actions. Messages and actions are further decomposed as events. An MSC specifies a partial order of the

execution of these events. MSCs can be composed to form larger MSCs through composition operations.

All MSCs defined for a system are usually collected together as an MSC document. The name of MSCs must be unique in the MSC document. So is the global variable name.

CHAPTER FOUR

4. STOCHASTIC MSC

In this chapter, we provide a way to extend the MSC formalism to include stochastic information. The extended MSC is called Stochastic MSC or SMSC. SMSC has more expressive power than MSC, and enables the performance analysis to be performed on the system, which is modeled as an SMSC.

4.1 Why Stochastic MSC?

The MSC formalism defined in the ITU (International Telecommunication Union) standard [3] is commonly used to specify the behavior of systems by constructing a series of MSCs. Each MSC is a description of a part of system behavior. The system-wide behavior description is achieved by combining these MSCs using the composition operators. But what kind of information about the system can we get given that the system is modeled as MSCs?

First, since an MSC describes a number of instances exchanging messages or performing some actions, we can know how many objects the system is made up of, what messages are exchanged, between which objects they are exchanged, and what actions are performed and by whom. Instances in an MSC actually represent objects of a real system.

Second, certain properties of system behavior can be specified. More precisely, the possible orderings in which actions and messages can occur are defined. An MSC not only contains entities for specifying system objects and their actions, but also imposes a partial order for the events that the system can engage in. We say that only a partial order

is implied because there can be events without a defined execution order. These events can happen in any order without violating any rules defined in the MSC formalism. A total order requires that all events can be ordered, directly or indirectly. This is not the case for MSCs. In a summary, MSCs tell us what the system is, what the system does, and how the system should do it. That is why MSC is a Specification Description Language (SDL).

The event ordering specified by MSCs is only one aspect of system behavior. Other properties regarding how well the system behaves, i.e. the performance of the system, cannot be ascertained from plain MSCs. This limitation is mainly due to the assumption made in the MSC formalism that all events are instantaneous. Under this assumption, MSC events cannot capture the characteristics of real system activities that do require time (or that have some relationship with time).

As a scenario description language, MSC is a good candidate for performance modeling since a performance model also describes the system behavior. In the paradigm of performance modeling, stochastic process theory is dominant. A system is first modeled as a stochastic process. The behavior of the system is assumed to be the same as the behavior of the stochastic process. A well-developed theory for stochastic processes can be used to analyze the system model and evaluate the system performance. Therefore, we relaxed the assumption in MSC formalism that all events are instantaneous and enable events to be associated with random time. The random time denotes the time required to complete the event. The new language is a stochastic extension to MSC. Thus, we call it Stochastic MSC (SMSC).

In the development of formal methods, there are many examples of extending a formalism to include stochastic time information. Petri Net was first defined without time information. Transitions in a PN are also instantaneous. Later, the PN formalism was extended to allow transitions to have time information. The new PN formalism was called Stochastic Petri Net (SPN). SPN models enabled the performance of the modeled system to be analyzed. SPN was further extended by allowing timed transitions to be mixed with instantaneous transitions. This extension to SPN is named Generalized SPN (GSPN). GSPN is expressively more powerful than SPN. But GSPN also has an extension: Stochastic Activity Network (SAN). SAN defines new constructs to build a model and further enhanced the expressive power. Another example would be Process Algebras (PA) and the corresponding Stochastic Process Algebras (SPA). PAs is used for analyzing system properties other than performance, while SPA is suitable for performance modeling. In fact, the MSC language has a formal notation based on PA[2]. Communicating Sequential Process (CSP) and CCS are two typical formalisms from the PA domain. PEPA (Performance Evaluation Process Algebra) is defined based on CCS. All PEPA activities must have exponentially distributed random time. As its name suggests, PEPA is defined for performance analysis.

4.2 Definition of SMSC

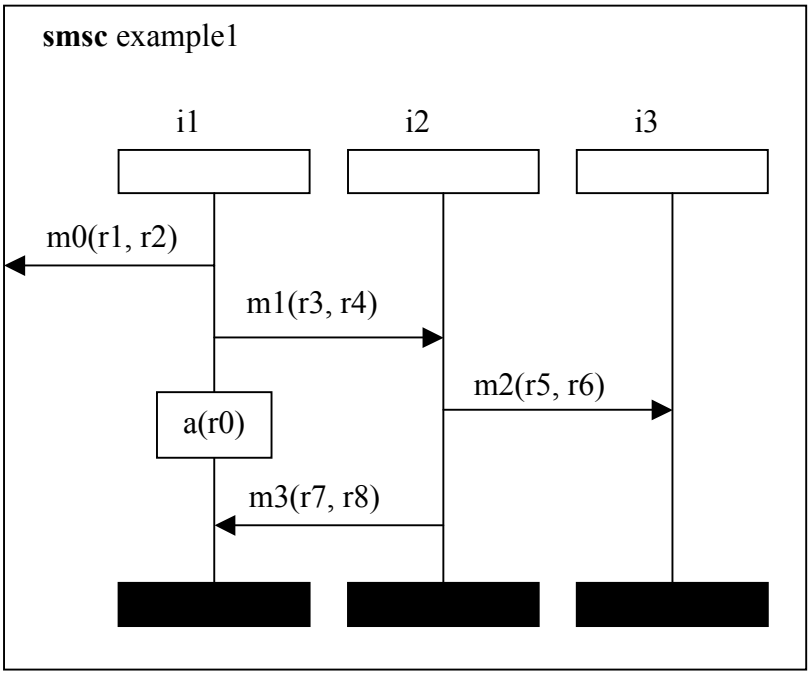
We define SMSC based on the language of MSC:

- **A Stochastic Message Sequence Chart is a Message Sequence Chart in which all events are enhanced to behave as activities by associating**

stochastic time information with them. The stochastic time associated with an activity defines the time needed to complete the activity.[2]

"Event" is usually used to describe the occurrence of something. When an event is associated with time, we call it an "activity." Activity means something that takes time to do.

The type of distribution of the stochastic time associated with activities can be deterministic, exponential, beta, etc. There is no restriction on what type of distribution a stochastic time can take. However, to simplify the description, we use the exponential distribution as the default distribution in the rest of this chapter. Figure 9 shows an example of an SMSC.



In the MSC language, there are two types of events: the events in message passing and the events for local actions. Hence, there are also two types of activities: message activities and local action activities or simply local activities.

[2] An immediate or instantaneous event is an activity associated with zero time.

A message in the SMSC language consists of two activities: the activity of sending the message and the activity of receiving it. Graphically, a message is represented by an arrow, which starts from the instance of sending the message and ends at the instance that receives the same message. A name is associated with a message and is followed by two parameters. The first parameter specifies the time for the sending activity and the second defines the time for the receiving activity. For example, message *m1* in Figure 9 has two parameters: *r3* and *r4*. *r3* specifies the rate of an exponentially distributed random variable that gives the amount of time needed to send the message. *r4* is for assigning the time to the activity receiving the message. Both r3 and r4 may be global variables so that their values can be easily modified later. The textural representation of messages is defined by adding a new keyword **withrate** to the MSC language as shown in Figure 10. Note that a new keyword **smsc** is defined to distinguish SMSC from MSC and is used in both the graphical and textural representations.

```
smsc example1;
i1: out m0 to env withrate r1;
i1: out m1 to i2 withrate r3;
i1: action a withrate r0;
i1: in m3 from i2 withrate r8;
i2: in m1 from i1 withrate r4;
i2: out m2 to i3 withrate r5;
i2: out m3 to i1 withrate r7;
i3: in m2 from i2 withrate r6;
endmsc;
```

Figure 10 Textual representation of SMSC

Local activities are also assigned random time in the same way as messages. But only one parameter is required.

4.3 Comparing MSC with SMSC

The SMSC language is different from the MSC in that SMSC activities are not instantaneous. Therefore, SMSC provides more information about a system than MSC. However, one may ask the question "Can SMSC provide the information regarding the modeled system that MSC provides?" or "Is the partial order of events imposed by MSCs still applicable to SMSC activities?" After comparing these two formalisms, we amazingly found that the answer is YES.

4.3.1 Constructs

All constructs (instances, messages, local actions, conditions, etc.) defined on MSC can be used for SMSC. The graphical representation of a SMSC looks the same as an MSC except for the additional parameters mandatory to activities in the SMSC.

As for textual representations, all the keywords defined in MSC are still defined on SMSC. Although new keywords are defined for SMSC, the method of describing SMSC is the same as that of MSC.

Most of the new keywords deal with the specification of random times for activities except for the keyword **smsc**, which denotes the MSC specified is actually an SMSC. For example, if an activity is associated with exponentially distributed random time, the keyword **withrate** is used in the description and is followed by a parameter that specifies the rate of the exponential distribution. We only need to provide one such parameter because the exponential distribution requires only one parameter. Other distributions may be specified by defining the corresponding keywords and providing the required parameters. In this thesis, we focus on the exponential distribution only.

SMSC and MSC have the same composition operators. SMSCs can be combined vertically, horizontally, or alternatively. The semantics of these composition methods in SMSC are identical to that of MSC.

High-level SMSC (HMSC) is defined in the same way as HMSC. HSMSC organizes SMSC references using the same nodes defined on HMSC. The interpretation of the organization is done in a similar way as what is defined for HMSC.

4.3.2 Ordering Rules

SMSC has different ordering rules. Under the new ordering rules, a SMSC imposes a partial order on its activities. This partial order is the same as that imposed by an MSC.

The two assumptions made in MSC are for precisely ordering events. The assumption of instantaneous events is obvious. If events can last for a period of time, it would be quit possible that another events starts before an already stated event finishes. In this case, what is the order of these two events? The assumption that no two events can be executed at the same time means any two events have a specific order. An event either happens before or after the other one. Hence, the execution of events forms a trace that describes the system behavior.

In SMSC, we relax the first assumption. As a result, the second assumption can no longer be held and is also relaxed.

We have mentioned that activities cannot be ordered. But if we decompose an activity into two events, one for the starting of the activity and the other for the ending of it, we will find a new way to order activities. The order of activities can be defined as either the

order of starting events or that of the ending events. By this definition, the order of activities may not be unique for an execution of these activities.

Since instances are independent in SMSC, activities are executed concurrently. Even if the starting times are different, two activities may finish at the same time because the execution time is a random variable. Therefore, it is possible that two events happen at the same time. If two events happen at the same time, they are treated as if they can be in any order.

We will show later that these ambiguities in ordering activity events will not prevent us from defining the partial order the same as that defined in MSC.

There are five ordering rules regarding the ordering of activities and activity events:

1) *The event of starting an activity must happen before the event of finishing the same activity.*

2) *Activities attached to an instance are executed sequentially in the same order as they are given on the vertical axis from top to bottom. An activity can only start after the previous one finished.*

3) *The activity of sending a message must finish before the activity of receiving the same message can start.*

4) *Activities in a coregion can happen in any order, but their execution must abide by rule 1.*

5) *If general orderings are used, they are treated as messages in terms of ordering these activities. In other words, the activity pointed to by a general*

ordering symbol can only start after the activity from which the general ordering originates has finished.

The first rule describes how to order the two events (start and finish) in an activity. Obviously, the starting event should always happen before the ending event. The second rule covers the ordering of activity events associated with the same instance. If each activity is treated as two consecutive events, the ordering of these events is the same as that defined for MSC.

The third rule is for ordering events in a message. The order of activities of different instances can be derived from this rule. A message includes two activities, and hence four events: the event of starting to send the message, the event of starting to receive the message, the event of finishing the sending of the message, and the event of finishing the receiving of the message. The precise restriction for their order is that the event of starting to send a message must happen before the event of starting to receive the message, and the event of finishing the receiving of the message must happen after the event of finishing the sending of the message. In other words, a message must be sent before it can be received, and the sending of the message must have finished before the receiving of it can finish. However, we define a stricter rule: the sending of a message must have finished before the receiving of it can start. This rule is to prevent a message from being completely received before the end of sending the message has not occurred.

The fourth and fifth rules are defined for ordering events in a coregion or for being controlled by general orderings. The interpretation is easy to understand.

Under these ordering rules, whether using the order of starting events or the order of ending events as the order of activities, this order imposed by an SMSC is sure to comply

with the partial order imposed by the corresponding MSC if the time information is removed from the SMSC. Therefore, an SMSC imposes the same partial order on its activities as an MSC does on its events. This result is mainly due to the strict ordering rules defined for messages and general orderings in SMSC.

Although we may have two different orderings for activities' starting events and ending events, both of the orderings will comply with the partial order imposed by the corresponding MSC. Any two activities that can be orders differently must correspond to the events that have undefined order in the corresponding MSC.

4.3.3 Traces vs. Processes

An MSC specifies a set of valid traces that the system can take. If we define the sequence of activities as a trace, an SMSC specifies a set of valid traces the same as an MSC. In addition, an SMSC also specifies a stochastic process.

The main difference between the MSC and SMSC languages is that SMSC defines a stochastic process while MSC does not. SMSC can describe the system behavior more precisely than MSC by providing users with more information about the system. The stochastic process enables users to do performance analysis about the system. This is the reason that we extend MSC to SMSC.

4.4 The Underlying Stochastic Process

To show that an SMSC defines a stochastic process, we use an indirect way. It is known that a Stochastic Activity Network defines a stochastic process [33]. It can be shown that a SMSC is equivalent to a SAN, hence a SMSC also defines a stochastic process.

4.4.1 SAN

SAN is an extension to GSPN. In addition to the common constructs defined for GSPN: places, directed arcs, and transitions. SAN defines two new elements: input gates, and output gates. The transitions are called activities in SAN.

Input gates are used to control the enabling of activities. Associated with each input gate are a predicate and a function. An activity can only be enabled if all its input gate predicates evaluate "true." The input gate function defines the marking change if the activity to which it connects fires. An input gate must be connected to all places whose markings affect or are affected by the firing of the activity. The graphical symbol for an input gate is a triangle: ▷ .

An output gate contains a function, which defines the marking change if the activity to which it connects fires. The symbol for the output gate is also a triangle: ▷. To distinguish an input gate from an output gate, the arc that starts at one vertex of the input gate triangle always ends at an activity. For an output gate, the arc will end at a place. For example, the triangle in the left part of Figure 11. is an input gate, and the right one is an output gate.

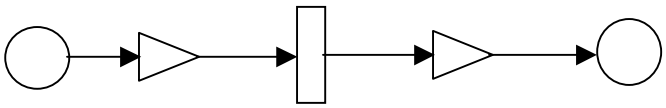


Figure 11 Gates in a SAN

Gates enable the modeler to manipulate markings and control the activities in a more flexible way. Hence SANs is expressively more power than GSPNs.

4.4.2 SMSC to SAN

A SMSC can be translated into a SAN. They specify the same underlying stochastic process.

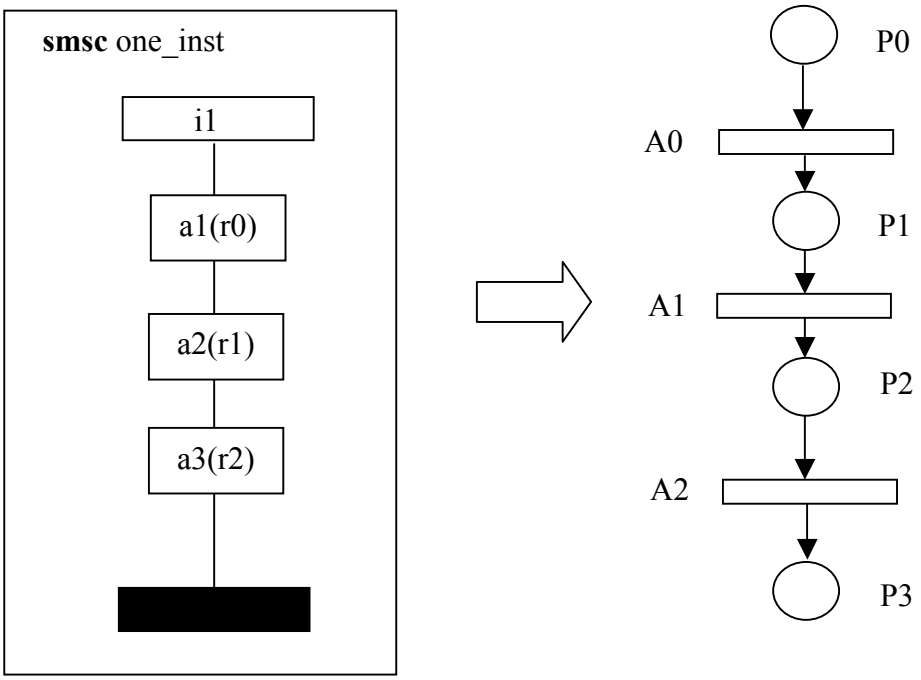


Figure 12 Translating an instance to a SAN

First, let us look at an independent instance with no message exchange. This instance just performs a series of local activities. Such an instance specifies a sequential process if no coregion is defined along the instance axis. The translation from a trivial instance to a SAN is trivial as well. Local activities are translated as SAN activities. Between any two consecutive activities a place is added. Also, we add one place before the first activity and another one after the last activity. This method of translation is illustrated in Figure 12.

The SMSC with one instance and three local activities is translated to a SAN with four places and three activities. Each place can at most have one token. In fact, only one of

these places will contain a token at any time. The place that has a token denotes the current state of the SAN or the state of the corresponding instance.

Next, a more complex example is discussed. In this example, there are two instances between them there are two messages. Each instance also performs a local activity. This example is shown in Figure 13.

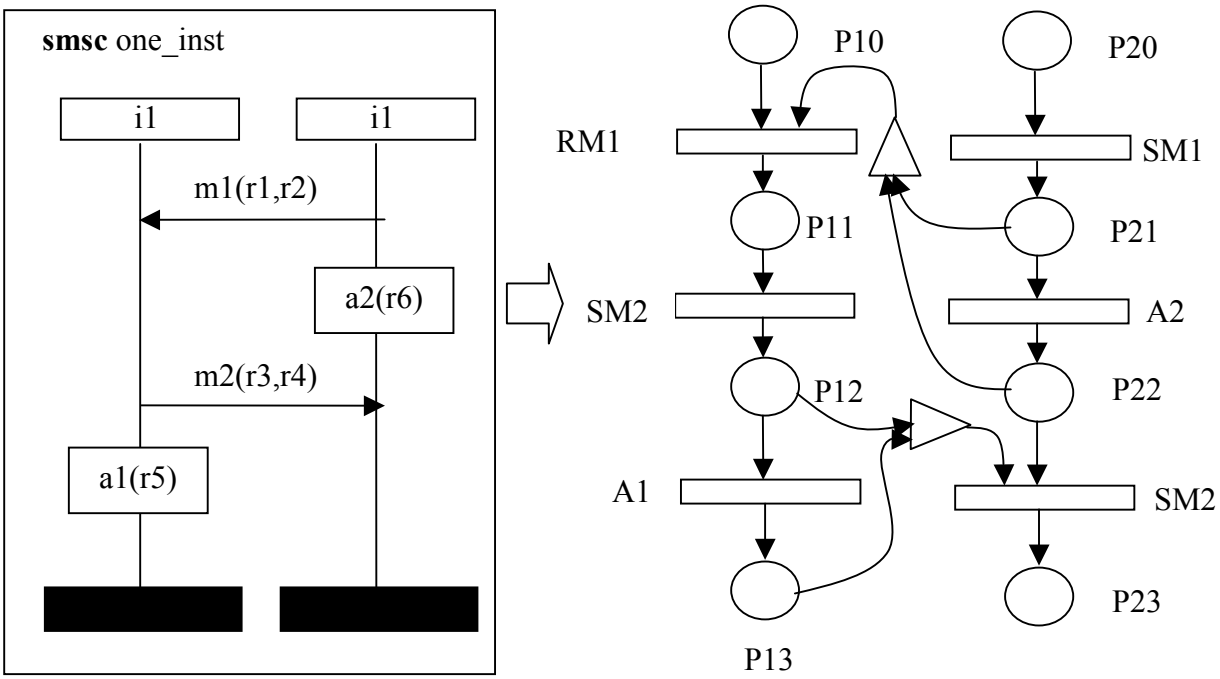


Figure 13 Translation of an SMSC to a SAN

If we look at the instances individually, each instance also specifies a sequential process. However, the condition that an activity can be executed in one process depends on the state of the other one. Message exchanges impose new restrictions on the enabling of activities.

According to the third ordering rule in section 4.3.2, the activity of receiving a message can only be executed after the activity of sending the same message has finished its execution. This means the activity of receiving a message can only be enabled after

the execution of the instance that sends the message has past a point, which denotes the end of sending the message. In the SAN model in Figure 13, activity RM1 represents the activity of receiving message *m1*, and SM1 denotes the activity of sending message *m1*. RM1 can only be enabled if there is a token in P21, P22 or P23. A token in either place means the sending of the message has finished. This restriction is modeled by the addition of an input gate. Note that the place P23 is excluded because there is activity before P23 that receives a message from the instance that executes RM1. Before finishing RM1, the instance cannot start SM2, and hence activity RM2 cannot be executed and it is impossible to have a token in P23. The predicate of the input gate should evaluate "true" if there is a token in either P21 or P22. Similarly, the enabling of RM2 is also controlled by an input gate that depends on the markings of P12 and P13. Of course, the enabling of an activity for receiving message also depends on the state of the instance that performs the receiving activity.

A general ordering between two activities from different instances imposes the same restrictions on the enabling of the later activity as a message does. The general orderings defined on activities of the same instance have no special meanings except for activities in a coregion. If a coregion is specified, activities in the coregion can run concurrently. Before all the activities in a coregion have finished execution, the activity which follows the coregion cannot be executed. If a general ordering is specified between two activities in a coregion, these two activities are executed sequentially as specified by the general ordering.

Any SMSC can be translated to an equivalent SAN. The execution order of activities imposed by an SMSC is preserved in its SAN equivalence. Therefore, a SMSC defines a stochastic process equivalent to the one defined by its corresponding SAN translation.

4.4.3 The Difference Between SMSC and SAN

Although SMSC models can be translated into SAN models, SMSC language is different from SAN in several aspects.

First, the purposed of modeling a system in SAN is to analyze the system performance. While a SMSC model not only enables performance analysis, but also describes the system behavior in terms of a specification description language. Therefore, SAN only cares about the internal dynamic behavior of the system and models the system as abstract as possible. A SAN model is usually complicated, and one is hard to capture the profile system to be modeled. SMSC can provide a clear overview about the modeled system.

Second, SAN and SMSC use different components. The basic components in SAN are activities, places, and input/output gates. But SMSC uses messages, local activities, and instances as its basic components. The concept of message is unique to SMSC in that messages imply the execution order of activities between different instances.

Finally, SMSC imposes a partial order on the execution of activities by carefully defining the ordering rules. There is no ordering rule defined on SAN actvitites. Although we can use SAN to mimic the behavior specified by the SMSC as we did in the previous examples, addition input gates have to be included in the SAN model. Since we need one extra input gate for any message and the input gate must connect to all the subsequent

places, too many input gates could be introduced for large models and the resulting SAN model would be a very complicated one comparing with the SMSC model.

Therefore, SMSC can be used to specify a system more clearly and concisely than SAN. SMSC also enables the performance analysis to be conduced on the system model just as a SAN model does.

4.5 Summary

In this chapter, we defined a new formalism Stochastic Message Sequence Chart based on the MSC language. SMSC is an extension to MSC. SMSC can be used to describe the system behavior in the same way as MSC does. However, SMSC includes stochastic time information and is capable of performance analysis, which cannot be done with just MSC.

The following chapter introduces a way to analyze SMSC models.

CHAPTER FIVE

5. MAPPING SMSC TO MÖBIUS

Now that we have defined the SMSC language and it is capable of performance modeling, we need to provide a tool to analyze the SMSC models. Instead of creating a new tool for solving SMSC models, we decide to integrate the SMSC formalism into the Möbius framework and use the Möbius tool to solve SMSC models. Since the Möbius tool supports multi-formalism modeling, building SMSC into the Möbius framework not only provides a tool for solving SMSC models, but also enables SMSC model to interact with models from other formalisms. In this chapter, we study the theoretical possibility of adding SMSC into the Möbius, and also give suggestions about how and what is needed for implementation.

5.1 Motivations and Problem Definition

To analyze an SMSC model, we can use one of the following three ways:

1) Develop a tool specifically for solving SMSC models.
2) Develop a parser to translate SMSC to SAN and use UltraSAN to solve the corresponding SAN model.
3) Integrate SMSC into the Möbius framework and use the Möbius tool to analyze the SMSC model.

We reject the first two methods and decide to adopt the third method due to the following reasons.

First, the Möbius tool provides discrete event simulators and analytical solvers that are capable of solving any models within the Möbius framework. Once a new formalism is integrated in the framework, the existing solvers are ready to solve models expressed in the new formalism. It is not necessary to develop solvers for the new formalism. All we need to do is to express our models using the Möbius entities.

Second, SAN has been integrated into the Möbius framework. In fact, the Möbius tool borrowed lots of ideas from the UltraSAN tool, such as model replication, performance variable specification, study editor, etc. The solvers available in the UltraSAN tool are also available in the Möbius tool. We do not need to translate SMSC to SAN if we can use Möbius entities to describe the SMSC models.

Finally, the most important advantage that we build SMSC formalism into the Möbius framework is that the SMSC formalism can be used for multi-formalism modeling. SMSC models can be easily joined with models from other formalisms (available within the Möbius tool) and form large heterogeneous models. Integrating the SMSC formalism into the Möbius framework enables the SMSC formalism to use the full features of the Möbius toolkit.

The Möbius framework defines three basic entities: state variables, action, and action groups. These basic entities are the building blocks of any model. In addition, an abstract functional interface (AFI) is also defined. The AFI can be used by other models or solvers to access the model information or to control the execution of the model. These basic entities and the interface has been implement as base C++ classes in the Möbius tool.

The Möbius framework requires that any formalism in the Möbius must implement the AFI and describes its model based on these basic entities. To build the SMSC formalism into the Möbius tool would require that SMSCs be decomposed into a set of state variables and a set of actions. Groups are not used when we describe the SMSC models. The state change and the ordering of action firings are determined by the structure of the SMSC model.

Therefore, before we can use the Möbius tool to solve a SMSC, the following three problems must be solved:

1) How to define SMSC states and the corresponding state variables.
2) How to define SMSC actions.
3) How to organize state variables and actions to represent the same model structure as defined in the SMSC.

The following three sections will answer these questions.

5.2 Identifying State Variables in SMSCs

To define the state of an SMSC, we must examine the components to see that the SMSC contains what necessary information for specifying the state of the system. An SMSC contains a number of independent instances. The instances send messages to each other and/or perform some local activities. SMSC may contain conditions that govern the execution of some activities. Local activities can also perform operations on local or global data. These constructs are used to model a system and contain the information that describes the system state.

5.2.1 Instance state

In section 4.4.2 we have shown that an SMSC can be translated into a SAN. The given example also showed that an instance with three activities corresponds to a SAN with four places. Places in the SAN model represent the system states. This implies that instances do have states.

The state of an instance should reflect which activity has been executed. Since an instance specifies a sequential execution order of its activities, it is important to keep the information about the execution of activities so as to ensure the sequential order. Initially, the instance is in a state that no activity has been executed. After executing the first activity, the state of the instance evolves to a new state that reflects the fact that the first activity has been executed. This process goes on until the last state has been reached, which shows all activities have finished.

The number of states that an instance can have depends on the number of activities associated with the instance. First, if an instance has no coregion defined on it, the number of states is given by the following equation:

$$NumInstanceStates = NumInstanceActivites + 1 \tag{5.1}$$

where *NumInstanceStates* is the number of states, and *NumInstacneActivites* denotes the number of activities on the instance.

We have two methods of representing the instance states. One method is to define a Boolean variable for each state. This method comes from the SAN equivalence of a SMSC. We have said in section 4.4.2 that each place in the SAN model can have at most one token, so a Boolean variable can be used to represent the state. But we reject this

method because the number of variables would be too many if an instance has a large number of activities attached. Actually we can use only one variable to hold the state information.

An instance that has no coregion specifies a strict sequential process. Activities can only be executed in the order they are given from top to bottom along the vertical instance axis. The execution of a later activity implies that all previous activities have finished. Therefore we can represent the instance state using an integer variable that holds the value of how many activities have been executed. Initially, the value is 0, meaning no activity is executed. The value increments by 1 after each activity is executed. From the value of this variable, we can immediately know which activity has finished and which activity is the next one to execute. It gives us no less information than a large number of Boolean variables. Furthermore, it uses less memory and is easy to manage. As long as the number of activities is within the rang of integer values (this is always the case), the state of an instance can be kept simply by using an integer variable.

Second, if a coregion exists in an instance, equation (5.1) no longer holds. Activities in a coregion can be executed in any order. A coregion brings additional states to the instance. To represent the state of a coregion, we have to associate each activity in the coregion with a Boolean variable. The "true" value denotes the finish of the execution of the activity, while the "false" value denotes the activity has not been started. The number of additional states brought by a coregion is at most

$$2^{NumCoregionActivities} \qquad (5.2).$$

If we exclude the coregion activities from the instance activities, equation 5.1 can be used to calculate the number of instance states. The total number of the states is the sum of this number and the number of states contributed by the coregion. Finally, if more than one coregion appears in an instance. Each coregion contributes at most the number of additional states given by (5.2).

5.2.2 Conditions

As defined in the MSC language, conditions represent system state. Therefore, conditions are good candidates for state variables. Depending on how many states a condition represents, the type of the state variable for a condition can be either Boolean or integer.

5.2.3 Data

SMSC can also perform operations on data just as MSC does. Data defined on SMSC are also state variables. The change of the data value represents the state change of the model. The type of the state variable for a data member is the same as the type of the data member.

5.2.4 Special Entities

Some special entities are defined in the MSC language. They are capable of sending or receiving messages. Theses entities include the **environment**, **lost** and **found**. Messages can be sent to or received from the **environment**. There is no order defined on environment. Therefore, we cannot consider the environment as an instance. Messages that are sent but not received by an instance are called incomplete messages. Incomplete

messages are considered to be directed to an entity: **lost**. Similarly, a found message is the one that no instance sends and is considered to originate from an entity: **found**.

To represent these special entities in the Möbius framework, we define one state variable for each. The state of the **environment** may contain the number of messages sent and received. So we can define a structure that contains two integers which represent the type of state variable for the entity **environment**. The state of **lost** can be used to count how many messages are lost. Thus, an integer is used to represent its state. The state of found is actually fixed. It must act as if the sending of the message has finished and enable the activity of receiving the found message.

5.2.5 Shareable vs. Non-shareable State Variables

The Möbius framework uses the concept of state sharing to join models from the same or different formalisms. If a state variable is shared with other models, the value of the state variable can be changed by other models too. The change of value represents the state change. Therefore, the behavior of the model is affected by the behavior of other models.

Not all the state variables we defined are shareable. For example, if the state variable defined for an instance is shared with other models, the increase of the state variable's value by other models may cause some actions to be considered finished even though they have not been executed. This is referred as state jump. Whether the state jumps ahead or back, the sequential execution order will be disturbed. Therefore, state variables from instances are not shareable.

Conditions and data will not affect the sequential order and hence these state variables are shareable. There is no need to share the special state variables for **environment**, **lost** and **found** because they are special state variables used only for SMSC.

5.3 Identifying Actions in SMSCs

By definition in the Möbius framework, actions are the only entities that can change the system state by changing the values of state variables. Thus any components in SMSC that can change the value of state variables will give us actions. These components include local activities, message activities, and setting conditions. Although data operations change the value of state variables that represent the data, data operations are not considered as actions because they are not components of SMSC. Data operations are performed by local activities or message activities.

5.3.1 Local Activities

Local activities can perform data operations and the completion of an activity must also increment the state variable that represents the instance to which the activity is attached. Thus, local activities are Möbius actions. If data operations are defined on the local activity, the execution of this local activity must also change the state variable representing the data. The execution time distribution for the action coming from a local activity takes the same distribution function as that of the local activity.

5.3.2 Message Activities

A message consists of two activities. The sending activity is performed by the instance that sends the message, and the receiving activity is performed by the one that receives the same message. Data operations can also be defined for message exchange. When the

activity of sending the message completes, it must adjust the state variable to reflect the fact that the message has been sent. Likewise, the completion of receiving a message should change the state of the instance that receives the message. Therefore, a message can be represented by two Möbius actions.

5.3.3 Setting Conditions

Conditions have two forms: setting conditions and guarding conditions. Setting conditions set the system to some particular state. Guarding conditions control the system behavior by restricting the execution of certain activities.

The setting conditions are Möbius actions since they change the system state.

The following figure (Figure 14) shows an example of an SMSC and its corresponding

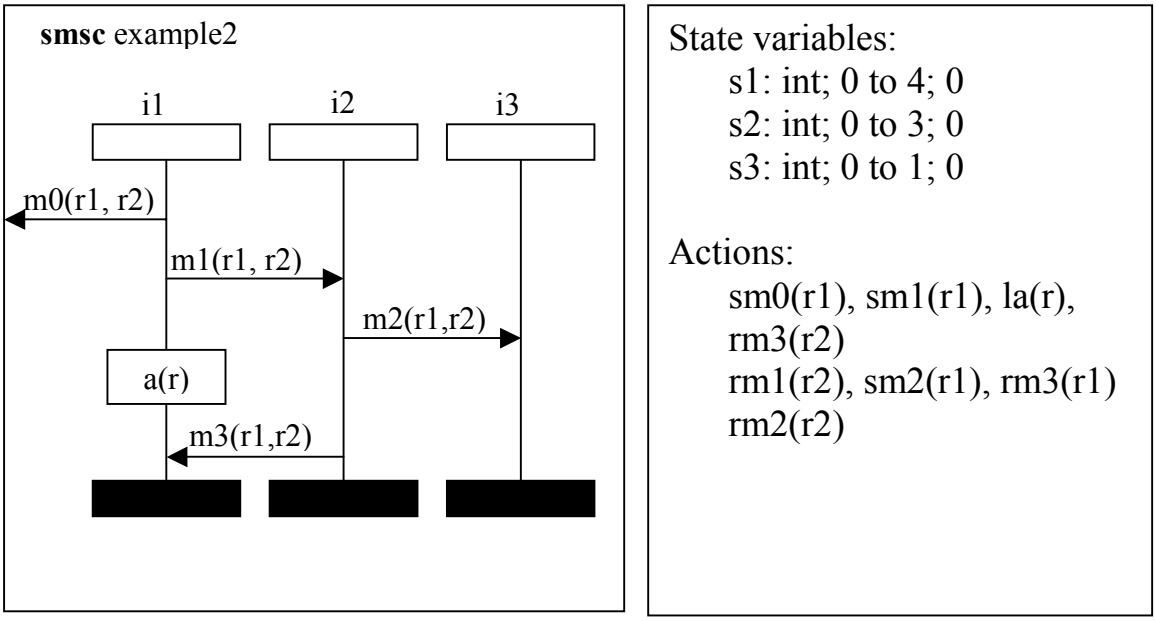


Figure 14 State variables and actions from an SMSC

state variables and actions. Action *rm1* corresponds to the activity of sending the message *m1*, and *sm1* corresponds to the receiving of message *m1*. Action *la* is for the local

activity *a.* The same naming rules apply to other action names. The state variables *s1, s2 and s3* represent the state of instances *i1, i2, and i3*, respectively.

In summary, the SMSC constructs and their corresponding Möbius entities are shown in Table 7.

Table 7 Mapping SMSC constructs to Möbius entities

SMSC Constructs	*Möbius Entities*
Instances	State Variables
Messages	Actions
Local Activities	Actions
Conditions	State Variables
Setting Conditions	Actions
Data	State Variables
Special Components (env, lost, and found)	State Variables
General Orderings	Taken care of by Actions

5.4 Expressing SMSC Models in Möbius Framework

To express SMSC in Möbius, we must define state variables and actions. State variables represent the model state. Actions can change the state variables' value and hence the state of the model. Since SMSC imposes a partial order on the execution of activities, the firing of actions must comply with this partial order. Therefore, these state variables and actions must be organized in a way that the partial order is ensured.

5.4.1 Deriving SMSC State Variable Classes

Based on the Möbius BaseStateVariableClass (see Appendix B), we can derive state variables classes for SMSC models. These state variable classes include SMSCInstanceClass, SMSCSharabelStateVariableClass.

SMSCInstanceClass must contain all the information necessary to describe an instance including its state, its coregion, activities associated with it, and especially the order of the activities. The data members defined on the SMSCInstanceClass are shown in Table 8.

Table 8 Data members defined on SMSCInstanceClass

Data Members	*Descriptions*
int Current;	The number of activities that has been executed sequentially
BOOL **CoregionState	Point to an array of Boolean variables whose true value means the corresponding activity has finished;
int NumCoregin;	The number of coregions defined on this instance;
struct {int start; int end} *coregion;	Point to a series a coregion structures that defines the starting and ending of coregions by the sequence number of the activities staring from 0.
ActivityClass *Activities	A list of activities in the order as they are given in the SMSC.

In general, the data member **Current** reflects the current state of the instance by maintaining a value that equals to the number of activities that have been executed. The value of the data member **Current** increases by 1 after each activity finished execution. For example, a value 3 means the first three activities have finished execution. But if the last finished activity (suppose it is the number 6 activity) is in a coregion, this number does not necessarily mean that the first 6 activities have finished. The finish of an activity in a coregion is recorded in a separate array: **CoregionState**. One must further refer to the **CoregionState** array to find out whether the number 6 activity has finished.

Another important data member is **Activities**, which maintains a list of activities defined on the instance. The order of the activities, together with the value of **Current**, is used to determine which activity can be enabled and which activities have finished execution.

As we have pointed out, instance state variables are not shareable. We need to define some sharable state variables so that SMSC models can be joined with other models. SMSCSharableStateVariableClass are used to define shareable state variables. Since their types could be integer, Boolean, or a structured type, it would be better to define them as template classes. A template class can take type as a parameter when it is instantiated. Therefore, Within the class definition, we define a pointer that points to the state variable's value. This is important when sharing this state variable with others because these shared state variables can point to the same memory location that stores the current state variable's value. A template class can be defined as:

```
template <Class T> Class SMSCSharableStateVariableClass{
T *state; // point to the state value.
}
```

5.4.2 Deriving SMSC Activity Classes

Activity class is derived form the Möbius BaseActionClass (see Appendix B). Although there are three different activities in SMSC: local activity, message activity, and the activity of setting conditions, we only need to define one activity class.

Two important properties regarding an activity are under what condition it is enabled and what state change it causes after it is executed. The activity class must contain

information necessary to specify its enabling condition and its firing effect. For a local activity, it can only be enabled if the one that precedes it has finished. For message activities, the sending activity's enabling condition is the same as a local activity. While the enabling of the receiving activity depends on not only the previous activity of the same instance but also the state of the sending activity of another instance. Only after those two activities have finished can the receiving activity be enabled. The setting condition activity has the same restriction as a local activity. Therefore it is not necessary to distinguish message activities from local activities or setting condition activities if we include enough information in the SMSCActivityClass.

The SMSCActivityClass is defined with two new data members in addition to those defined on the BaseActionClass. Their meanings are described in Table 9.

Table 9 Data members defined on SMSCActivityClass

Data Members	*Descriptions*
SMSCActivityClass *previous;	Point to a list of activities that must have been executed in order to enable this activity;
SMSCInstanceClass *MyInstance;	Point to the instance this activity is attached to.

General orderings imposes same restriction on activities as messages. The effect of general orderings can be taken into account using the same idea for messages.

The firing of an activity will change the instance state variable, may change a condition state variable if it is a setting condition activity, and may perform data operations and change data state variables.

5.4.3 Deriving SMSC Model Class

The SMSCModelClass is derived from the Möbius BaseModelClass (see Appendix B). The SMSCModelClass is used to organize the state variables and activities for a SMSC model. In addition to what is defined in the BaseModelClass, the SMSCModelClass contains additional data members as shown in Table 10.

Table 10 Data members defined on SMSCModelClass

Data Members	Descriptions
SMSCInstantClass *SMSCInstances;	All instances in a SMSC
SMSCActivityClass * SMSCActvities;	All activities in a SMSC
SMSCModelClass **NextSMSCs;	SMSC Models following this model
Short NumSMSCs;	The number of SMSCs in the NextSMSC list;
BOOL Enabled;	The current SMSC is enabled if "true"

The structural information of a SMSC is kept in the SMSCActivityClass and SMSCInstanceClass rather than the SMSCModelClass. The SMSCModelClass only acts as a container of state variables and activities. In addition, the SMSCModelCalss also provides methods of composing two or more SMSC models.

One additional data member defined on the SMSCModelClass is the Boolean variable **Enabled. Enabled** is used as the guarding condition for all the activities in the SMSC. Activities in an SMSC can only be enabled when this variable is set to TRUE.

5.4.4 Model Composition

The default methods of combining two models by joining the shared state variable in the Möbius framework may not work when combining two SMSCs. The reason is that SMSCs impose a partial sequential order on the execution of activities. SMSCs can be

joined vertically, horizontally, or alternatively. This is beyond what can be expressed by the Möbius joining operations. But the Möbius joining operation is valid when joining SMSCs with other models through shareable state variables for the purpose of forming the Möbius composed models.

The SMSC formalism defines its own model composition methods. We can use SMSCModelClass to specify these compositions. First, the vertical composition is achieved by setting **NextSMSCs** to point to the next SMSC and **NumSMSC** to 1. This setting means there is one SMSC that immediately follows the current SMSC, and after all activities of this SMSC have finished, the execution continues to the next SMSC that is specified by the pointer: **NextMSCs**.

For alternative composition, there are two or more SMSCs following the current SMSC. The number of SMSCs that follow this SMSC is stored in the data member NumSMSCs. The variable NextMSCs is a pointer array of which each pointer points to a SMSC that follows the current SMSC. Probabilities may be assigned to each subsequent SMSC to determine which one should be executed after the execution of the current SMSC.

Horizontal composition cannot be specified using this mechanism because it involves changing the common instance into a coregion. This requires significant changes to the structure of the current SMSC. If horizontal composition is specified, it must be resolved before using an SMSCModelClass variable to represent it.

5.5 Solving SMSC Models

Once the SMSC models are described using the Classes derived from the Möbius base classes. It is relatively easy to analyze it. The Möbius built-in solvers can be used to solve SMSC models.

If all activities are associated with exponentially distributed random time, the underlying process is a Markov process. The Möbius analytical solvers can be used to quickly solve the model. Before using any analytical solvers, the state space must be explicitly generated. This implies that the model has to have finite states and if so, then the Möbius tool provides the utility to generate the state space.

The Möbius simulators can be used to solve any model regardless of the type of distribution associated with activities. If the underlying process is not Markov, then discrete event simulators are the only choice when solving the model for performance measures. Before solving the model, performance variables must be defined for measuring the desired system properties.

5.6 Summary

In this chapter, we provide a way to define actions and state variables for the SMSC models, and also give the requirement in deriving the corresponding C++ classes. Some model composition methods are discussed as to how thay can be easily implemented. SMSC is well suited for integration into the Möbius framework and provides a *new* atomic modeling formalism for Möbius users.

The next chapter will provide an example to show how SMSC can be used with other formalisms to model a system.

CHAPTER SIX

6. A NETWORK COMMUNICATION EXAMPLE

In this chapter, we provide an example to illustrate that SMSC models can be joined with models from other formalisms, such as SANs, through equivalence sharing. SMSC formalism provides a new type of atomic models in the Möbius framework. The heterogeneous model can be solved using the Möbius solvers.

6.1 A Communication System

We consider a simple system with two computers connected through a cable. The processes running on one computer send files to those running on another computer. The communication protocol used by the data link layer is the stop and wait protocol[34].

The sending process first opens a file for transmission. The data in the file is then broken into small data blocks and each block corresponds to a frame. The frame is the smallest data block to transmit. Data blocks are then handed to a process that creates a frame and stores the frame into a sending buffer. Whenever there is a frame in the sending buffer, the sending process will try to send the frame over to the other computer using the stop and wait protocol.

The receiving process is the inverse of the sending process. A received frame is kept in a receiving buffer. If the frame is correctly received, it will be handed up to a data block buffer. After all the data blocks have been received, they will be combined into a file. The sending and receiving processes are molded as SANs. The stop and wait protocol is modeled as an SMSC.

6.2 Model the Stop and Wait Protocol

The stop and wait protocol is the simplest communication protocol that can coordinate the communication between two entities that run at different speeds and have limited buffer space. The sender sends out a data block and then waits for the receiver to acknowledge the receipt of the data. Before the sender gets the acknowledgement, it cannot start sending the next block of data. This is necessary to prevent a fast sender from flooding the slow receiver if the receiver has limited receiving buffers.

If the stop and wait protocol is used on an unreliable channel, i.e., data in transmission may be damaged due to errors that occur in the channel, then the technique of retransmission must be adopted. The sender starts a timer after it transmits a data block. If the timer goes off before it receives the acknowledgement, the data is considered lost and the sender retransmits the same data block. Upon receiving a data block, the receiver first checks if the data is correct. If correct, then the receiver sends back a positive acknowledgement. Otherwise, a negative acknowledgement is sent back. Note that the receiver may receive duplicated data if the acknowledgement is lost. In our example system, we assume an unreliable channel is used.

To model the stop and wait protocol, we need four SMSCs. Each of them describes a scenario for the behavior of this protocol. The four scenarios are shown using SMSCs in Figure 15.

The first SMSC shown in Figure 15 (a) represents the success of the data exchange. The data is correctly received and so is the acknowledgement. No data got lost in the channel. Figure 15 (b) describes the scenario where an error occurred during the

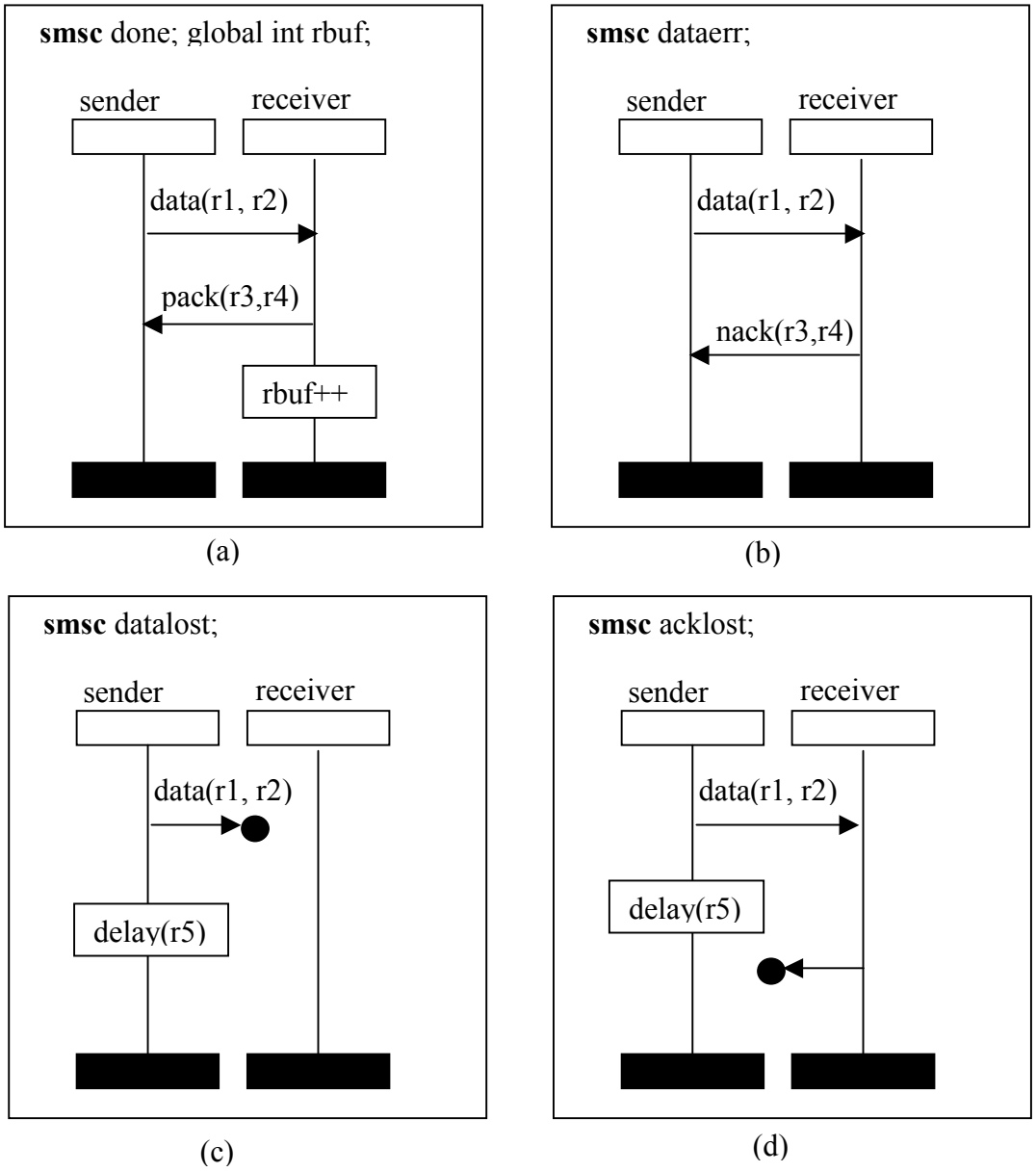


Figure 15 The 4 scenarios of the Stop and Wait protocol

transmission. In this case, a negative acknowledgement is sent back. The scenario shown in Figure 15 (c) happens if the data is completely lost in the channel. The receiver did not

receive anything at all. So it can perform no action. The sender has to resend the data after a period of time specified by the delay activity. The delay activity is used to simulate a timer. Figure 15 (d) represents the scenario that an acknowledgement is lost. Since the sender did not receive the acknowledgement, it will resend the data after some time.

Figure 16 provides an additional SMSC, **GetFrame**, in order to specify how the sender gets data from the sending buffer. This SMSC serves as the starter for the stop and

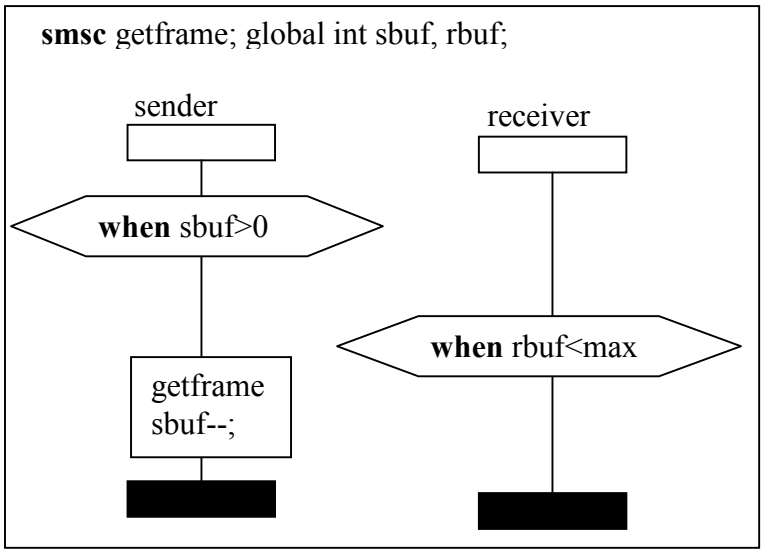


Figure 16 The GetFrame SMSC

wait protocol. The full behavior of this protocol can be described by combining these five SMSCs. Figure 17 shows the composition methods. The GetFrame SMSC describes the behavior of the sender when it fetching a data frame from the sending buffer. After a data frame is acquired, the execution proceeds into one of the alternative four scenarios. The SMSC **done** represents the success of data exchange. If **done** is chosen and has finished, the execution goes back to GetFrame. The SMSCs **done** and **GetFrame** form a loop. If

done is not selected as the follower of **GetFrame** in this execution, the execution has to loop among the four scenarios indefinitely until the SMSC **done** is selected.

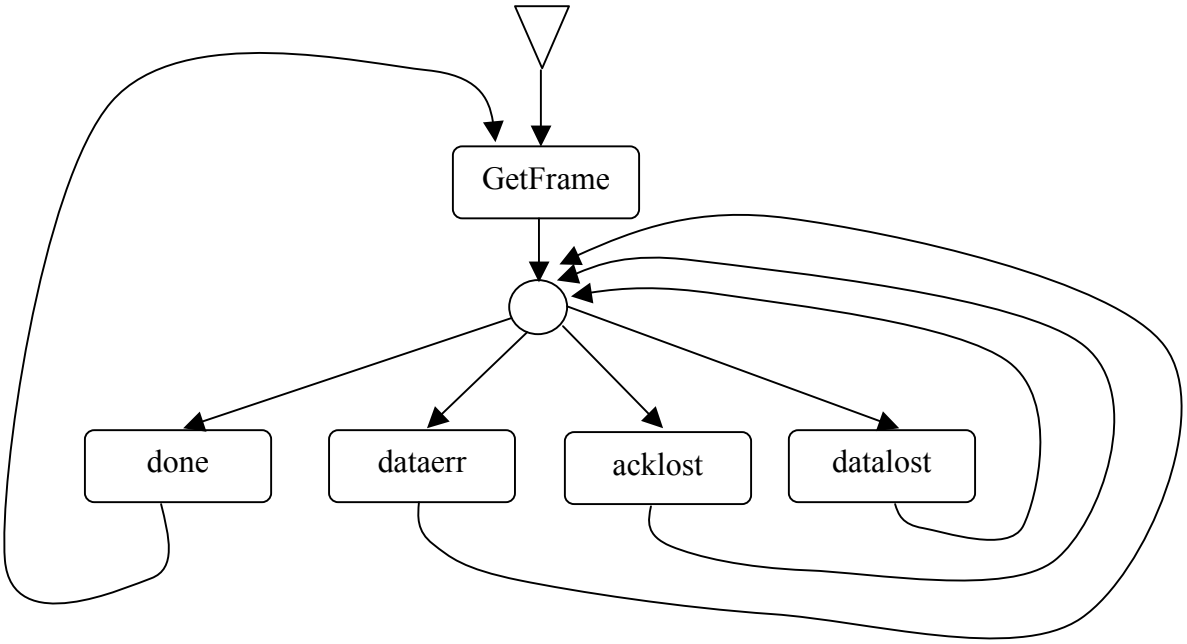


Figure 17 The model of the Stop and Wait protocol

6.3 Modeling the Data Sending and Receiving Processes

The data sending and receiving processes are modeled as Stochastic Activity Networks. The SAN model for the sender is shown in Figure 18.

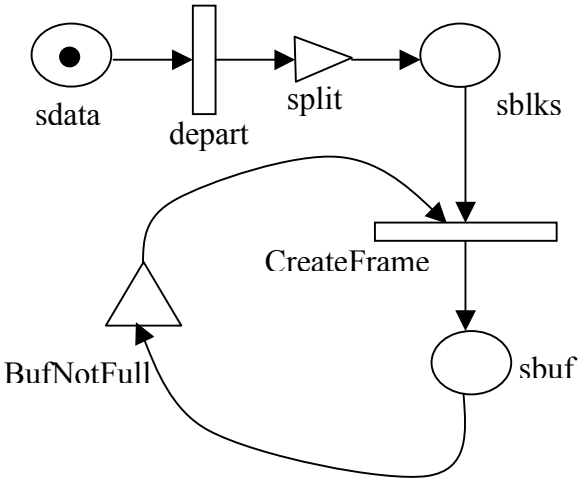


Figure 18 The SAN of the sender

The data sending process or the sender works in this way. A token in the place *sdata* represents a large block of data, for example a file, is ready to transmit. The SAN activity *depart* fires, and the output gate *split* defines the number of tokens that are put into the place *sblks*, which represents the block buffer of the sender. The SAN activity *CreateFrame* can fire if at least one token exists in *sblks* and the predicate of the input gate *BufNotFull* evaluates to true. This predicate is true if the sending buffer is not full. Each time *CreateFrame* fires, a token is dropped into the place *sbuf.* Each token in *sbuf* represents a data frame that will be sent using the stop and wait protocol. *subf* represents the sending buffer.

The SAN model for the data receiving process or the receiver is shown in Figure 19.

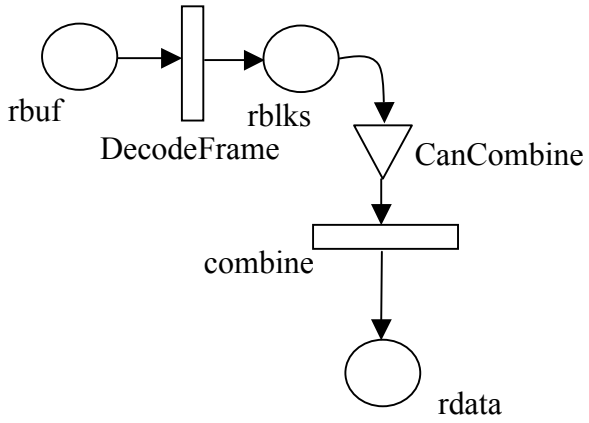


Figure 19 The SAN of the receiver

The procedure of processing the received frames is the inverse of what is done by the sending process. Whenever there is a token in the place *rbuf*, the SAN activity *DecodeFrame* will fire and deposit a token in the place *rblks.* When the number of tokens accumulated reaches a certain value, the input gate that controls the enabling of the SAN activity *combine* may evaluate true on its predicate. Then, *combine* fires and a token is

put in the place *rdata.* This token represents the same large block of data as the one in the place *sdata.*

6.4 A Heterogeneous Model of the Whole System

The heterogeneous model can be constructed using the Möbius Join and Replicate mechanism as shown in Figure 20.

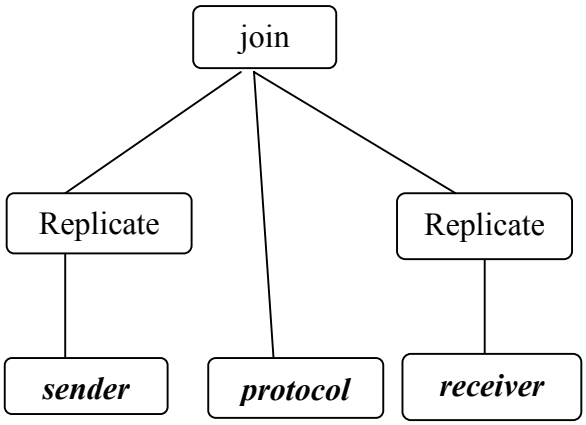


Figure 20 Construct the system model

In Figure 20, ***sender*** and ***receiver*** refer to the SAN models of the sender and receiver. ***protocol*** refers to the SMSC model of the stop and wait protocol. The sender and receiver models may be duplicated several times so that the behavior of a system with several senders and receivers can be studied without building a complicated model in which the sender and receiver models are drawn several times.

Before the models are joined, we must specify the shared state variables. The Join construct in Möbius uses the shared state variable to join different models together, whether they are from the same formalism or different formalisms. In our example, *rbuf* and *sbuf* are shared state variables. In the SAN model, places *rbuf* and *sbuf* are defined as state variables in the Möbius representation. The global data *rbuf* and *sbuf* in the SMSC

are also defined as state variables. These state variables are shareable. In fact, they represent the same system components in different models. The number of tokens in the place *sbuf* of the SAN model can be seen by the SMSC model when it checks its global data s*buf*. The decrement of s*buf* in SMSC model means the removal of a token from the place s*buf* in the SAN model. The increment of the global data *rbuf* in the SMSC model will be interpreted by the SAN model as a token in put into its place *rbuf*. Through these shared state variables, the SAN model and the SMSC model can affect the behavior of each other. The behavior of the whole system is described by models from both formalisms.

6.5 Summary

In this chapter, we described a simple communication system including two computers. Processes running on one computer send data to another computer through a cable. The communication protocol used here is the stop and wait protocol.

The stop and wait protocol are described by four scenarios and modeled as SMSCs. The data sending process and data receiving process are modeled as SANs. The SAN model and SMSC model are connected together using the Möbius Join and Replicate techniques. Shared state variables are defined in both types of models.

This result shows that the SMSC formalism is able to interact with models from other formalisms and that the Möbius tool can solve the SMSC models.

CHAPTER SEVEN

7. CONCLUSIONS AND FUTURE STUDY

The Message Sequence Chart formalism and the Möbius multiple modeling framework were studied. Based on the MSC formalism, we defined a new formalism – Stochastic Message Sequence Chart, which is an extension to the MSC formalism. SMSC can be used to describe the system behavior in the same way as the MSC language. Furthermore, SMSC models contain more information regarding the system than their corresponding MSC models. By associating with each activity a stochastic execution time, the SMSC models specify an underlying stochastic process. System performance measures that cannot be derived from MSC models can be studied by using SMSC models. In this sense, the SMSC language is more powerful than the MSC language.

The possibility of integrating the SMSC formalism into the Möbius framework was investigated. On the basis of this investigation, we discovered that the SMSC formalism can be well fitted into the Möbius framework. The key issue for building the SMSC formalism into the Möbius framework is to specify the SMSC models using the Möbius entities: actions and state variables. We defined the SMSC state variables and SMSC activities, which correspond to the Möbius state variables and actions, respectively. The structural information of the SMSC model is retained when the model is specified in Möbius framework. We also provide the primitive requirement about how to define some C++ classes that are used to specify SMSC models. Some of the model composition methods specified in the SMSC formalism can be realized using the C++ classes, namely, vertical composition and alternative composition. Loop is a special vertical composition and is also realizable within the Möbius framework.

The next step in this work would be to implement the SMSC formalism into the Möbius framework. This requires the implementor to collaborate with the Möbius group at University of Illinois at Urbana-Champaign. As another formalisms in the Möbius framework, SMSC will provide a user interface and the interface should be implemented in Java in order to make it platform neutral. The front-end user interface will enable users to specify SMSC models in the Möbius tool. Eventually, the graphical or textural SMSC models are translated to C++ source files, which are further complied and linked with the Möbius C++ libraries to generate an executable model which is either simulated or solved analytically.

Some constructs of the SMSC language, including inline expressions, horizontal compositions, and SMSC references, have not been defined within the Möbius framework. Further research will reveal how this can be accomplished.

Another area of future work is to define the action-sharing method for SMSC. Instead of sharing state variables, an SMSC model may be composed with other models by sharing activities/actions.

BIBLIOGRAPHY

1. F. Sheldon, G. Xie, O. Pilskalns, and Z. Zhou, *A Review of Some Rigorous Software Design and Analysis Tools.* Software Focus Journal, 2002. 2(4): p. 140-149.

2. ITU-T, *Formal Semantics of Message Sequence Charts*. 1998: Geneva.

3. ITU-T, *Recommendation Z.120: Message Sequence Chart(MSC)*. 1999: Geneva.

4. B.R. Haverkort and I.G. Niemegeers, *Performability modelling tools and techniques.* Performance evaluation, 1996. 25(1): p. 17 (24 pages).

5. M.K. Molloy, *Performance Analysis Using Stochastic Petri Nets.* IEEE Transactions on Computers, 1982. C-31(9): p. 913-917.

6. T. Murata, *Petri Nets: Properties, Analysis and Applications.* Proceedings of the IEEE, 1989. 77(4): p. 541-580.

7. J.C.M. Baeten, *Process algebra: special issue editorial.* The Computer journal, 1994. 37(5): p. 474.

8. G. Ciardo, Muppala J., and Trivedi, K.S. *SPNP: Stochastic Petri Net Package*. in *3rd Int'l Wkshp on Petri Nets and Performance Models*. 1989. Kyoto, Japan: IEEE Computer Society Press, Los Alamitos, CA.

9. J. Hilston and H.U. Hermanns, *Stochastic Process Algebras: Integrating Qualitative and Quantitative Modeling*. 1994, Univ. of Erlangen-Nurnberg: Germany.

10. E.L. Cunter, A. Muscholl, and D.A. Peled, *Compositional Message Sequence Charts.* Lecture Notes in Computer Science, 2001(2031): p. 496-511.

11. W. Damm and D. Harel, *LSCs: Breathing Life into Message Sequence Charts.* Formal Methods in System Design, 2001. 19(1): p. 45-80.

12. D. Daly, Doyle, and Stillman, Webster, Möbius: An Extensible Framework For Performance and Dependability Modeling, 1999. www.crhc.uiuc.edu/PERFORM

13. W. Sanders, Obal, W., Qureshi, A., and Widjanarko, F., *The UltraSAN Modeling Environment.* Performance Evaluation, 1995. 24(1): p. 89-115.

14. J. Hillston, *A Compositional Approach to Performance Modelling*. 1996: Cambridge University Press.

15. G. Clark and W.H. Sanders. *Implementing a Stochastic Process Algebra within the Möbius Modeling Framework*. in *Process Algebra and Probabilistic Methods: Performance Modelling and Verification: Proceedings of the Joint International Workshop, PAPM-PROBMIV 2001*. 2001. RWTH Aachen, Germany: Berlin: Springer.

16. Z. Zhou and F. Sheldon. *Integrating the CSP Formalism into the Mobius Framework for Performability Analysis*. in *Proceedings of PMCCS'5*. 2001. Erlangen Germany, Springer-Verlag.

17. J.F. Meyer, *Performability: A Retrospective and Some Pointers to the Future.* Performance Evaluation, 1992. 14(3-4): p. 139-156.

18. Petri Nets World, 2002. http://www.daimi.au.dk/PetriNets/tools/

19. B.R. Haverkort, *Performability Evaluation of Fault-tolerant Computer Systems Using DyQNtool+.* International Journal of Reliability, Quality, and Safety Engineering, 1995. 2(4): p. 383-404.

20. 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, 1995. 25: p. 117-135.

21. M. Veran and D. Potier, *QNAP2: A Portable Environment for Queuing System Modeling.* Modeling Techniques and Tools for Computer Performance Evaluation, 1985: p. 25-63.

22. C.H. Sauer and E.A. MacNair, *Simulation of Computer Communication Systems*. 1983: Prentice-Hall.

23. K.C. Chang, R.F. Gordon, P.G. Loewner, and E.A. MacNair, *The Research Queueing Package Modeling Environment (RESQME)*. 1993, IBM Research Yorktown: New York.

24. S. Gilmore and J. Hillston. *The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling*. in *Proceedings of the Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. 1994. Vienna: Springer-Verlag.

25. *UltraSAN User's Manual: Version 3.0*. 1995, Center for Reliable and High-Performance Computing Coordinated Science Laboratory, University of Illinois at Urbana-Champaign.

26. 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*. 1992: G. Balbo.

27. 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,*. 1996.

28. 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*. 1998. Berlin.

29. 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),*. 1995. Berlin: Springer.

30. 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*. 1996: Boston, MA.

31. R. Sahner and K.S. Trivedi, *Reliability Modeling Using SHARPE*. 1986, Computer Science, Duke University: Durham, NC.

32. G. Ciardo and A.S. Miner. *SMART: Simulation and Markovian Analyzer for Reliability and Timing*. in *Proceeding of IEEE International Computer Performance and Dependability Symposium (IPDS'96)*. 1996.

33. W.H. Sanders and L.M. Malhis, *Dependability Evaluation Using Composed SAN-Based Reward Models.* Journal of Parallel and Distributed Computing, no. 15, pp. 238-254, 1992.

34. A.S. Tanenbaum, *Computer Networks*. 3rd ed. 1996: Prentice-Hall.

APPENDIX

A. MÖBIUS GROUPS

A group is a collection of actions and/or groups that have a specific execution policy. The actions and groups contained in a group are called the members of the group. Actions in a group can compete or cooperate in a specialized way. In other words, a group can reduce the number of states in which a member action can fire. In a given model state, as a member of a group, an enabled action may not fire depending on whether it is chosen to represent the group. Only the group representative member can fire at any given state. Therefore, groups allow a formalism to implement non-race-based execution policy among a subset of actions. Groups control the execution policy of their members by implementing a selection process, which is an algorithm for determining which group member can fire at a given state.

A group defines a number of functions and data members. The **Members** is a set that contains groups and actions that are the members of this group. There is also a data member called **Actions**, which is the set of members of the group that are actions but groups. Similar to actions, the **Enabled** function defined on a group is a Boolean function to determine whether there is an enabled member in the group. The function **Enabled** returns TRUE if there is at least one enabled member in the group. The **Select** function defines how the individual member is selected from the set of enabled members. And the **ReslectPredicate** and **ReselctFunction** define when and how a group reselects another group member as its representative.

In implementing the group entity, a superset of functions defined on actions is implemented. Two additional functions implemented 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. If the selected member is a group, then the **Select** function is called in a recursive fashion on the selected group until the selected member is an action. The **Probability** function is used to calculate the value of the probability that an enabled member can fire in a given state. This function uses the action's rank and weight values, and is used by the **Select** function when selecting the representative of a group.

Groups are divided into two main subdivisions: preselection groups and postselection groups. Preselection groups choose their representative actions at activation time, whereas postselection group choose their representatives at firing time.

In the Möbius framework, preselection groups are further classified into two categories based on how the reselection conditions are defined. Reselection conditions are rules that define when a preselection group selects its representative member since there are many different time points at which a group can select its representative. The two categories are variable preselection groups and persistent preselection groups. A variable preselectoin group selects its representative whenever there has been a change in the enabling conditions of one or more group members, and there is at least one enabled member in the new state. The change of enabling conditions means 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 its representative in the state in which there is no enabled group member in the previous state, and at least one enabled

member in the current state. A persistent preselection group also reselects its representative when the representative member fires and the group is still enabled in the new state.

Postselection groups select their representatives at the firing time. For this selection policy to make sense, certain restrictions exist for group members. First, all member actions must have same firing time distribution. Thus, the Möbius tool can use any group member's distribution function to determine the scheduled time to completion of the group. Second, all member actions must have the same enabling conditions. This means that either all member actions are enabled or none of them is enabled. Finally, 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. In summary, all action functions of all members in a postselection group must be identical, except for **Ran**k, **Weigh**t, and **Fir**e.

As is said before, a group can have another group as its member. In this case, the parent is called group a multi-level group. Preselection groups and postselection groups are different in terms of creating multi-level groups.

Preselection can have any levels. The member groups of a preselection group can be preselction groups or alternatively postselection groups. When the top-level group chooses its representative, it calls its **Select** function. If the selected member is a group, its **Select** will be called. The process continues (as mentioned above) until an action is returned.

Preselection groups cannot be members of a postselection group because preselection groups do not have the same restrictions on their actions while postselection groups do. A postselection group may contain another postselection group as its member. But this implies actions in these two groups all have the same firing distribution, the same enabling condition, and identical reactivation function and predicate. It is better to organize these actions into one group rather than dividing them into two groups. Therefore, multi-level postselection groups are not necessary. Postselection groups are implemented to support only one level.

B. IMPLEMENTATION OF THE MÖBIUS TOOL

All the Möbius entities are implemented as C++ classes. These classes serve as base classes for all formalisms. Therefore, the implementation of these classes are not formalism-specific, instead, they are implemented in a more general way so as to provide general building blocks for formalism in Möbius. Each class also defines methods that provide a common interface through which solvers and models of different formalism can communicate with each other. For example, solvers built in the Möbius tool can communicate with the model by calling methods of the interface. The methods return generic information about the model. The same methods can also be used to exchange information between models specified in different formalisms.

Methods that require formalism-specific implementation are declared as pure virtual methods. In C++ terminology, pure virtual functions are functions that are declared on a parent class without any specific definition. The child class derived from the parent class is responsible to provide an exact definition. The base classes are called "the abstract classes" because the methods are not completely defined. Hence, every formalism must define its formalism-specific operations for methods of the abstract classes.

Each Möbius entity corresponds to a base class. Taking into account the base class for the Möbius model, there are totally 4 classes. They are BaseStateVaribleClass, BaseActionClass, BaseGroupClass, and BaseModelClass.

- ***BaseStateVariableClass***

BaseStateVariableClass defines methods and data members necessary to implement state variables. Methods defined on BaseStateVariableClass are summarized in Table 11. These methods are categorized into three types: those that deal with state variable's state, those that are used for state sharing, and those that manipulate the data members that store the list of actions affected by or affecting this state variable.

The state manipulation methods are defined as virtual functions. They are SetState, StateSize, CurrentState, and PrintState. The state change of a model is closely related to the formalism that specifies the model. The BaseStateVaribleClass has no specific definitions of these methods, but provides the description of what these methods intend to do. The formalism implementor is responsible to fulfill the requirements of the state manipulation methods. SetState is used to set the state of the state variable by copying data from a specific memory location pointed by a void pointer. A void pointer is able to point to any type of data. Hence, the formalism specific data type is not important in the definition of this method. Actually, SetState is only used when the entire model needs to be reset by solvers. The action-firing-related state change is formalism specific, and must be implemented in the derived class.

Table 11 Methods defined on BaseStateVariable Class

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 Listt<BaseActionClass>* getAffectingActions()	This method returns the affecting actions data structure
Const Listt<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(BaseAction Class*)	This method appends the specified action to the state variable's object SVAffectingActions
Void appendEnabledAction(BaseAction Class*)	This method appends the specified action to the state variable's SVEnabledActions object
Void copyAffectingActions(List<BaseAc tionClass>*)	This method copies the data structure passed in and uses it as its list of affecting actions
Void copyEnabledActions(List<BaseActi onClass>*)	This method copies the data structure passed in and uses it as its list of enabled actions
Void updateAffects(BaseStateVariableCl ass*)	This method will notify all the actions on the state variable's SVAffectingActions and SVEn-abledActions lists to inform them that this state variable is part of a sharing set

CurrentState writes the value of the state variable to a specific memory location. StateSize is used to determine how many bytes are needed to store the state variable state. PrintState displays the state variable's value on the standard output device of a computer. Usually, this is the screen.

State-sharing methods are used to access the state-sharing-related data members. These data members are usually Boolean variables indicating whether the state variable is shared, whether it is functionally shared, and if the state variable's state is store locally. For example, GetShared, GetFunctionallyShared, and GetStored. The method GetSharingCount returns the number of state variables that share state with this state variable. It returns 1 if the state variable is not shared.

BaseStateVariableClass contains data members that store the information about actions related to a state variable. Two lists of actions are defined in this class. SVEnabledActions is a list of all actions that are enabled by the state variable's value. SVAffectingActions contains all actions that their firing will affect the state of this state variable. These two data members are implement as list data structures that contain pointers pointing to the actual actions. The set of actions used to initialize these data structures for each state variable must be structurally determined from the model specification.

- ***BaseActionClass***

BaseActionClass is the implementation of the Möbius entity action. This implementation is quite straightforward. Table 12 shows the methods defined on this class and their corresponding description. Most of the methods are virtual functions because their exact definitions are determined by the formalism that implements them.

Table 12 Methods defined on BaseActionClass

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

In addition to these methods, BaseActionClass also defines important methods and data members that facilitate efficient analysis of the model. Among them, two data members are AffectedStateVariables and EnablingStateVariables. AffectedStateVariabes is a list of state variables whose states are affected by the firing of the action. EnablingStateVariables stores a set of state variables that are used to determine whether

the action is enabled. To initialize these two data members, the corresponding set of state variables must be deduced from the structure of the model. Four methods are defined to change the value of these data members. They are addEnablingSV, addAffectedSV, replaceEnablingSV, and replaceAffectedSV. The last two methods are used when a state variable is shared. In this case, the pointer stored in these data members might need to be replaced by a pointer pointing to another state variable, which is the head of the list of shared state variables.

Action attributes (shown in Table 13) include GroupID, ExecutionPolicy, ActionName, and DistributionType. GroupID specifies the group to which this action belongs. ExecutionPolicy take one value from RaceResamping, RaceEnabled, and RaceAge. It governs the behavior of the action when it is interrupted. DistributionType defines the probability distribution used for describing the action's firing time distribution. The supported types of distributions are listed in Table 14.

Table 13 Action attributes

Attribute Name	*Description*
int GroupID	The group to which the action directly 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.

Table 14 Supported distribution functions

Distribution Name	*Parameters*
Exponential	Rate
Deterministic	Val u e
Geometric	P
Weibull	α, β
Normal	μ, σ^2
Lognormal	μ, α^2
Erlang	m,β
Triangular	a,b,c
Gamma	α, β
Beta	$\alpha 1, \beta 1$
Uniform	UpperBound, LowerBound
Binomial	t, p
NegativeBinomial	s, p
HyperExponential	rate1, rate2, p

Table 15 Performance variable related data members

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 impulses 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.
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
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

BaseActionClass also defines several different data structures that are used to implement performance reward variables. When the action fires, the affected performance variables are updated. Table 15 shows the complete list of the performance variable related data structures and their corresponding meanings.

- ***BascGroupClass***

The C++ class BaseGroupClass is a derived class from the BaseActionClass. Since at any given model state only one action in a group can fire, the group functions as an action in that sense. Functions defined on actions are also the functions defined on groups. The formalism specific functions for actions are defined as virtual functions for groups too. In addition to these action functions, BaseGroupClass also contains methods and data members for manipulating group members. A complete list is shown in Table 16.

Group members are kept in two lists: ActionMembers and GroupMembers. ActionMembers contains the list of group members that are actions. GroupMemebers is a list of the members that are groups. The functions appendGroup and appendMembers are used to add members to a group. Both of them require a group pointer as the parameter. The method appendGroup makes the group, which is passed in as the method's parameter, as a member of this group in the group member list: GroupMembers. While appendMembers does not make that group as a member of this group, instead, it makes the members of that group as members of the current group.

The SelectAction method is used to select an action from the group members as the representative of the group. The algorithm uses the actions' rank and weight to determine which enabled action should be selected.

Table 16 Methods defined on BascGroupClass

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 ith member that is a group
BaseActionClass* getActionMember(int)	This method returns the ith member that is an action.
void printGroup()	This method hierarchically prints out a group's membership

- ***BaseModelClass***

Models act as containers of actions, state variables, and groups. The BaseModelClass defines methods that are used by solvers or other models to access the Möbius entities in a model. Table 17 shows all the methods defined in the BaseModelClass. These methods can be categorized as list methods, state methods, and composed model methods.

Table 17 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>*)	This returns a reference to all of the actions contained in a model
void listActions(char*, List <BaseActionClass>*)	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(BaseStateVariable Class*, 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

The list methods include listModels, listGroups, listActions, and listSVs. We can see by their names that they each return the corresponding entity set to solvers or other models.

The state methods (SetState, CurrentState, CompareState, and StateSize) are defined to perform operations on the model's state variables. SetState is used to set the model to a specific state based on the passed memory pointer. CurrentState returns the current model state to a memory location specified by a pointer. CompareState is used to compare two model states and see whether they are equivalent. It is worth pointing out that one must use this method to compare two model states. Comparing the memory data of two model states byte by byte is not a good way to check the equivalency of two model states. The reason is that the difference between memory data does not ensure that the two model states are not equivalent. Finally, StateSize returns the number of bytes needed to store the model state, which is similar to the definition in BaseStateVariableClass.

The composed model methods are necessary for building composed models through state sharing. When two models are joined together by sharing state variables, the shared state variables are said to be in the same sharing set. Every sharing set has one state variable declared as the leader. The method getMainSharedVariable returns a pointer of the leader. But the method getListOfSharedVariables returns the head of a linked list, from which all the members in the sharing set are accessible.

Besides these methods, BaseModelClass also defines data members. Most of them are used to summarize the entities contained in the model. Table 18 shows the defined data members.

Table 18 Data members 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

C. THE MOBIUS SOLVERS

- ***Discrete Event Simulation***

The Möbius tool has two discrete event simulators: a transient simulator and a steady-state simulator. The transient simulator is used to obtain transient measures, i.e., the measures at time *t*, given that $t < \infty$. The steady-state simulator uses batch means with the deletion of an initial transient to solve for steady-state instant-of-time reward variables. The estimated statistical properties for reward variables include mean, variance, and distribution. For mean and variance, confidence interval can be specified.

The advantages of using simulation are:

- Simulation is applicable to any models, regardless the action's time-to-completion distribution.
- Simulation does not require the generation of the entire state space.
- Simulation does not require the model have a finite state space.

However, simulation could take quite a long time if either the rare event problem arises, or higher accuracy is desired [25].

- ***Analytical Solvers***

The Möbius tool has incorporated 7 analytical solvers. They are Accumulated Reward Solver (ARS), Transient Solver (TRS), Adaptive Transient Solver (ATS), Direct Stead-State Solver (DSS), Iterative Steady-State Solver (ISS), Deterministic Iterative Steady-State Solver (DISS), and Advanced Iterative Steady-State Solver (ADISS).

How to choose an appropriate solver depends on the model type, reward variable type, and the desired measure. ARS is used for solving transient interval-of-time reward variables. It gives the accumulated reward, as well as the time-averaged accumulated reward over the interval. TRS and ATS solve for instant-of-time reward variables with $t < \infty$. All the steady-state solvers solve for instant-of-time reward variables with $t \to \infty$. They use different techniques. DISS should be used when there is at least one deterministic action in the model.

Before using any analytical solver, the state space of the model must be explicitly generated. This implies the model has to have a finite state space. Another restriction for using analytical solvers is that the model must imply a Markov or Semi-Markov process. In other words, actions' time-to-completion must have exponential distribution and there is at most (for semi-Markov) one action with deterministic distribution.