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

DECEMBER 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
December 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. The SMSC formalism is integrated into the Möbius framework and the Möbius default solvers are used to solve SMSC models. We defined the mappings from SMSC to Möbius entities and implemented the required C++ classes to describe SMSC models in the framework. Together with other formalisms in Möbius, SMSC can be used as building blocks 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 will have 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

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 Submitted:

4. Frederick Sheldon and Zhihe Zhou, *Extending Message Sequence Charts for Performance Analysis in Möbius,* Annals of Software Engineering: Distributed and Mobile Software Engineering
5. Frederick Sheldon and Zhihe Zhou. *SMSC: A Modeling Language for Performability Analysis*, ACM Symposium on Applied Computing, SAC 2003, March 9-12 2003, Melbourne, Florida, USA

LIST OF TABLES

LIST OF FIGURES

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 introduces the Message Sequence Chart formalism and the Möbius framework. 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 can be described by a High-level MSC.

The Möbius framework is based on Möbius entities and the AFI. The AFI contains a series of functions that are defined in the Möbius entities and must be implemented by all formalisms in the framework. The structure of the Möbius framework and the method of building and solving models using the Möbius tool is also described.

Chapter 3 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 4, we provide a method of integrating the SMSC formalism into the Möbius framework. The components of SMSC are analyzed to derive state variables and actions, which are the Möbius entities necessary to implement a formalism within the Möbius framework. The implementation of SMSC classes is discussed. We also analyze the

complexity of the SMSC models and the corresponding solving time for certain performance measures.

Chapter 5 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 6 concludes this thesis and provides future research directions regarding SMSC and the Möbius tool.

CHAPTER TWO

2. MSC AND MÖBIUS FRAMEWORK

This chapter covers the basics of MSC and Möbius.

2.1 Message Sequence Charts

The full specification of the Message Sequence Charts language can be found at [3]. Here, we briefly introduce the MSC formalism and provide some basic concepts that are necessary to understand our work. These concepts include the basic constructs of MSCs, event ordering rules, the composition of MSCs and High-level MSCs.

The MSC formalism describes a system using a series of charts, each of which specifies part of the system behavior. These charts are combined together to depict the whole system. Inside each chart, there are several independent instances that represent components of the system and these instances exchange messages and perform actions. MSCs are always placed within the context of some encompassing environment. Instances in MSCs can send messages to or receive messages from their environment. An MSC can be represented graphically or textually.

Figure 1 shows an example of a graphical representation of one Basic MSC (BMSC). 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.

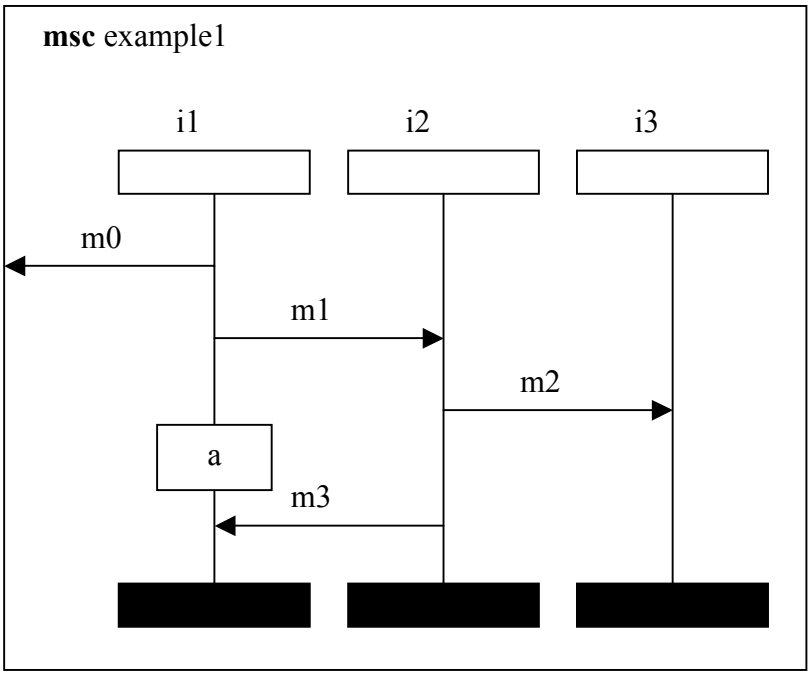

Figure 1 Graphical representation of an MSC

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**.

A BMSC describes a simple scenario and cannot have loops/branches in its instances. Loops/branches are always based on scenarios or BMSCs and specified by the way of composing BMSCs within the High-level MSC.

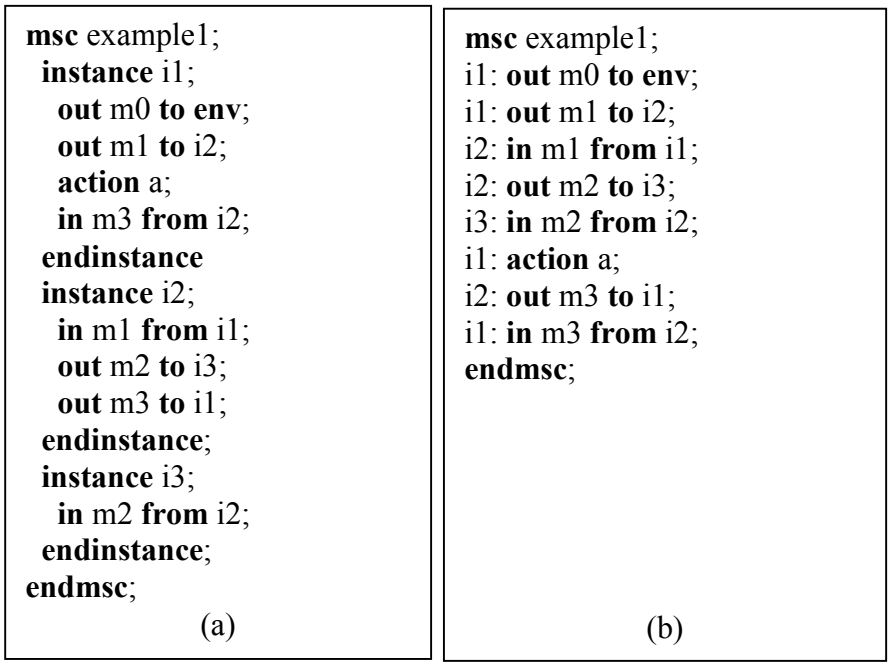

Figure 2 Textual representation of the example 1

A Message Sequence Chart is composed of interacting **instances**, which are the primary entities in an MSC. 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.

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. An arrow starting from the sending instance to the surrounding frame represents a message sent to the environment. If the message is sent but never consumed (i.e., lost), 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.

In addition to messages, **local actions** of an instance may be specified in MSCs. A local action describes an internal atomic activity of an instance and contains either an informal description of the activity, or a formal statement that defines operations on some data. Graphically, a local action is denoted by an action symbol on an instance with the action string inside (i.e., a box placed on and obscuring the instance axis). See local action *a* in Figure 1.

An MSC also specifies a partial order for the events inside the MSC using two basic ordering rules. The first rule concerns 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**. The second rule concerns the order imposed by messages. The key idea for defining this rule is that a message must be sent before it can be consumed. Therefore, the second rule is, **the event of sending a message must happen prior to the event of receiving the same message**.

To enable the description of unordered events along an instance axis, the MSC formalism introduces the **coregion** construct. A coregion is drawn as a dashed vertical line that replaces part of the instance axis. Events in a coregion may happen in any order. A **general ordering** is used to explicitly specify the ordering of two events whose ordering is otherwise undefined.

Message exchanges and local actions may be restricted by **conditions**. Conditions are an MSC construct that specifies the system states. There are **setting conditions** (i.e., set the system to certain state) and **guarding conditions**. A guarding condition precedes messages and local actions to further restrict their execution. When the condition holds, the message(s)/action(s) that follow the guard may execute.

The MSC formalism also supports structural design composed vertically and/or horizontally. Within one MSC, a reference can be used to refer to another MSC. Vertical composition connects the common instances that share the same name in two MSCs. Thus, the event execution trace of common instances in successive MSCs follows the execution of events in the preceding MSCs. While in horizontal composition, the events of common instances are interleaved. An MSC can have more than one MSC in vertical composition. In this case, the succeeding MSCs are alternatives of each other (i.e., called alternative composition).

Generally, the way to combine MSCs is to use a High-level MSC (HMSC), in which MSC references and other constructs are used to specify their composition. An HMSC cannot contain instances, messages or local actions although it can use conditions. HMSCs can only use MSC references because the goal

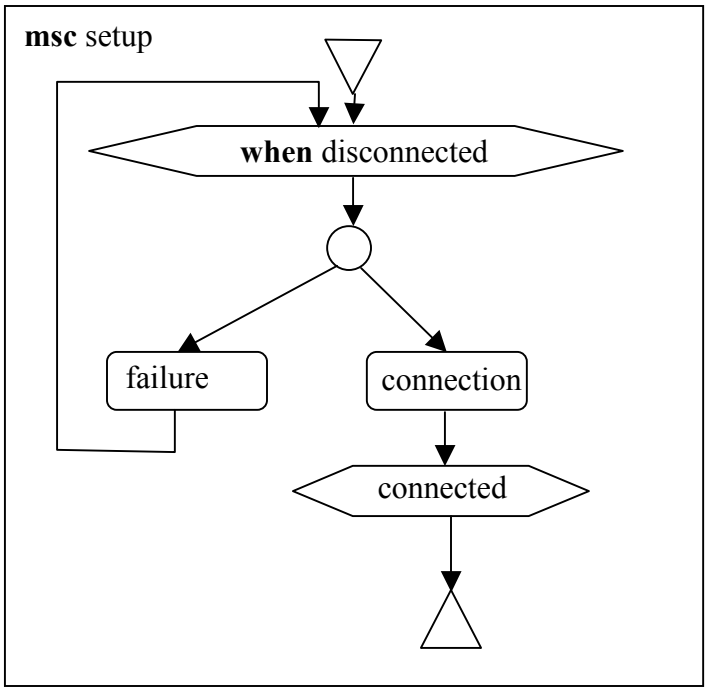

Figure 3. Example of High-level MSC.

of HMSC is to define how the basic MSCs are connected. Figure 3 shows an HMSC example.

2.2 Möbius Framework

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 [33]. The composition techniques developed permit models to interact with one another by sharing state, events, or results. This framework also supports multiple modeling languages and multiple model solution methods, including both simulation and analysis. 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 Möbius framework defines three basic Möbius entities: **state variables**, **actions**, and **action groups** (or **groups**). **State variables** hold the state of the model, or the state of the modeled system. The type of state variables could be a simple type such as integer, Boolean, or double, or a complicated structure type. The value of a state variable could also depend on the value of another state variable (i.e., the value of the state variable is a function of another state variable). **Actions** are the only Möbius entities that can change the values of state variables, thus the state of the model or the system. Actions could be instantaneous or timed. An instantaneous action takes no time, while a timed action is usually associated with some random times, which is called time-to-complete. Only after this period of time can the action complete or fire. Actions are enabled under certain system state and the firing of an action often changes the system state to a new state.

Groups contain one or more actions called group members. A group is enabled when at least one group member is enabled. However, not all enabled group members can fire. At any time, only one enabled group member is elected as the representative that can fire. Other enabled group members are ignored. The way to select its representative must be defined on a group.

All formalisms integrated into the Möbius framework must use the Möbius entities to specify their model. But this by no means implies different formalisms are translated into some universal modeling language because the Möbius entities are not fully defined modeling components. For example, actions can be enabled and can fire. But how to decide the enabling condition and what to do when they fire are left undefined. These issues are specific to the formalism and must be dealt with when implementing that formalism into the Möbius framework. The Möbius entities, together with the formalism specific information, are used to describe the model built on the formalism [34].

The Möbius framework defines an Abstract Functional Interface (AFI). The AFI is the core of the Möbius framework because it enables models to exchange information with other models and different solvers. The AFI also enables the Möbius solvers to solve a model without the knowledge of the underlying formalism. Thus, hybrid models that consist of models from different formalisms are solvable.

The Möbius AFI consists of functions that are implemented as C++ virtual methods within the implementation of the C++ classes for Möbius entities. Virtual methods can be redefined in the derived class so that the formalism specific behavior can be defined in terms of the Möbius entities for a given formalism. In the implementation of the Möbius tool, state variables are not simple variables, instead, they are implemented as an abstract

C++ class: BaseStateVariableClass. Actions and action groups are implemented as BaseActionClass and BaseGroupClass, respectively. One additional class BaseModelClass is defined as the container for Möbius entities. Each class contains several virtual methods that are part of the AFI. The virtual methods and their corresponding classes are summarized in Table 1.

Table 1 Möbius classes and AFI functions.

Class	AFI functions	Function description
BaseStateVaribleClass	int StateSize( )	Determine the number of bytes needed to store the state
	void SetState(void *p)	Sets the value of the state variable
	void CurrentState(void *p)	Writes the state to the memory location p.
BaseActionClass	Bool Enabled( )	Determines whether the action is enabled in the current model state
	double SampleDistribution( )	Returns the action's time-to-completion
	Fire( )	Defines how the action changes the state of the model
BaseGroupClass	(All functions defined in the BaseActionClass)	
	SelectAction ( )	Selects the action to represent the group
BaseModelClass	int StateSize( )	Determine the number of bytes needed to store the model state
	void SetState(void *p)	Sets the state of the model
	void CurrentState(void *p)	Writes the model state to the memory location p.
	void ListActions( )	Lists actions in the model
	void ListGroups( )	Lists groups in the model
	void ListSVs	Lists state variables in the model

The formalism in the Möbius framework must derive its own classes from these basic abstract classes and implement the AFI, i.e., provide their own implementation for those virtual methods.

The Möbius framework uses a hierarchical model construction method, as is shown in Figure 4. First, **atomic models** are built from single formalisms. Second, two or more atomic models form a **composed model** by sharing state variables. Then, reward variables are defined for atomic or composed models to form a **solvable model**. One or more solvable models, together with reward variables, can form a **connected model.** The solvable models are solved using the Möbius simulators or analytical/numerical solvers.

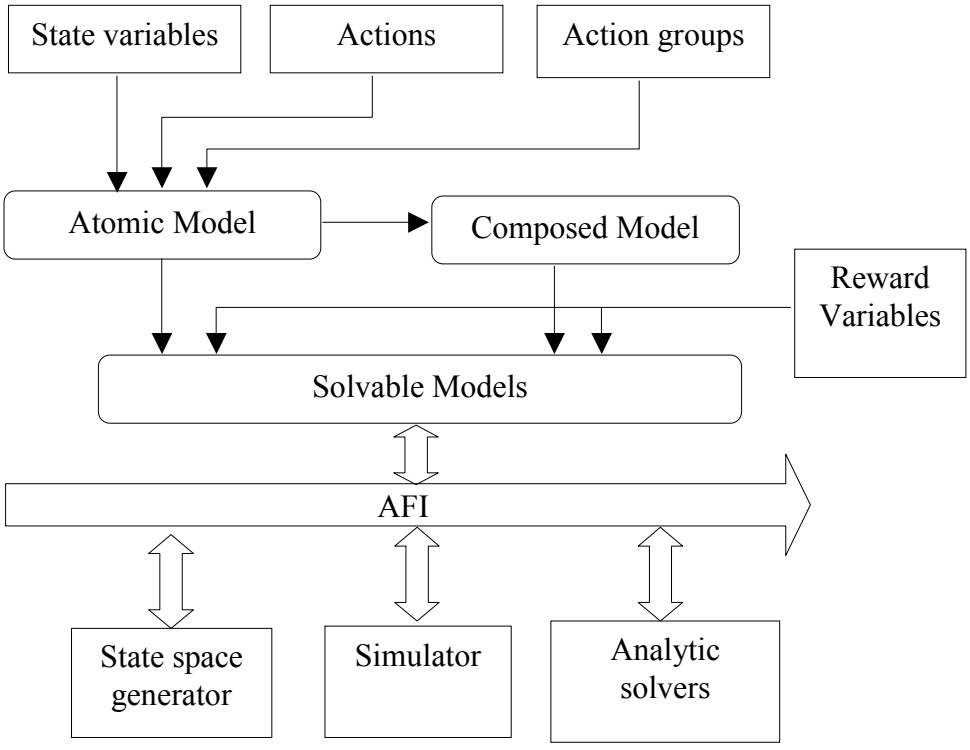

Figure 4 The Möbius framework.

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.

How to choose an appropriate solver depends on the model type, reward variable type, and the desired measure. 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].

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.

2.3 Summary

In this chapter, the MSC formalism and the Möbius framework were introduced. MSC 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, which are describe by an HMSC.

The Möbius framework facilitates composition of multiple heterogeneous models and multiple solution methods. The SMSC formalism is integrated into this framework and hence, the Möbius solvers can be used solve SMSC models.

CHAPTER THREE

3. 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.

3.1 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 a partial order is

implied because there could 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 a special case for MSCs in which all events have only one execution order. 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.

3.2 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 5 shows an example of an SMSC.

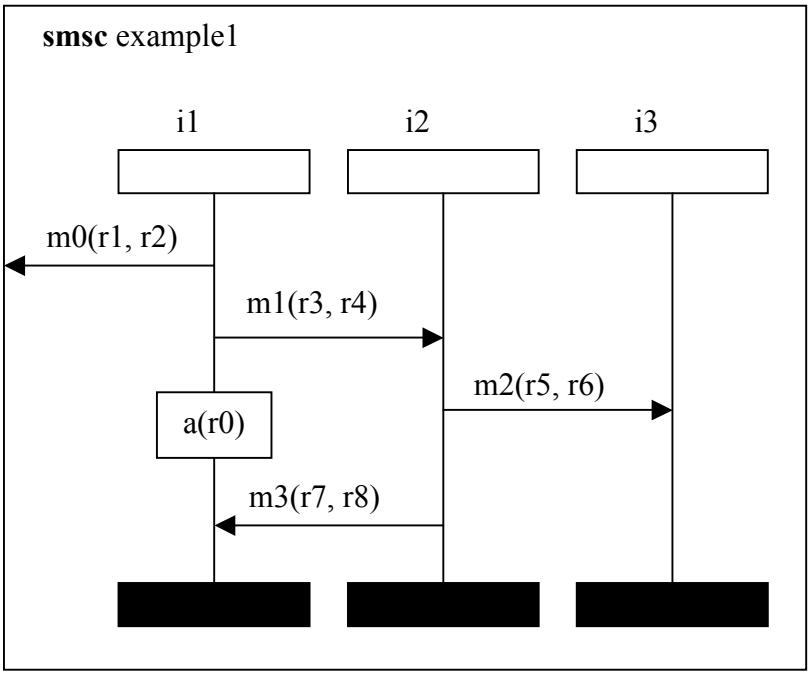

Figure 5 An SMSC example

In the MSC language, there are two types of events: the events in message passing and

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

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

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 5 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 6. 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 6 Textual representation of SMSC

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

3.3 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.

3.3.1 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.

3.3.2 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 quite possible that another event 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.

3.3.3 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.

3.4 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 [35]. It can be shown that a SMSC is equivalent to a SAN, hence a SMSC also defines a stochastic process.

3.4.1 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 7. is an input gate, and the right one is an output gate.

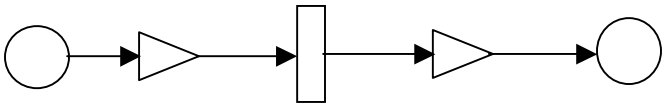

Figure 7 Gates in a SAN

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

3.4.2 SMSC to SAN

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

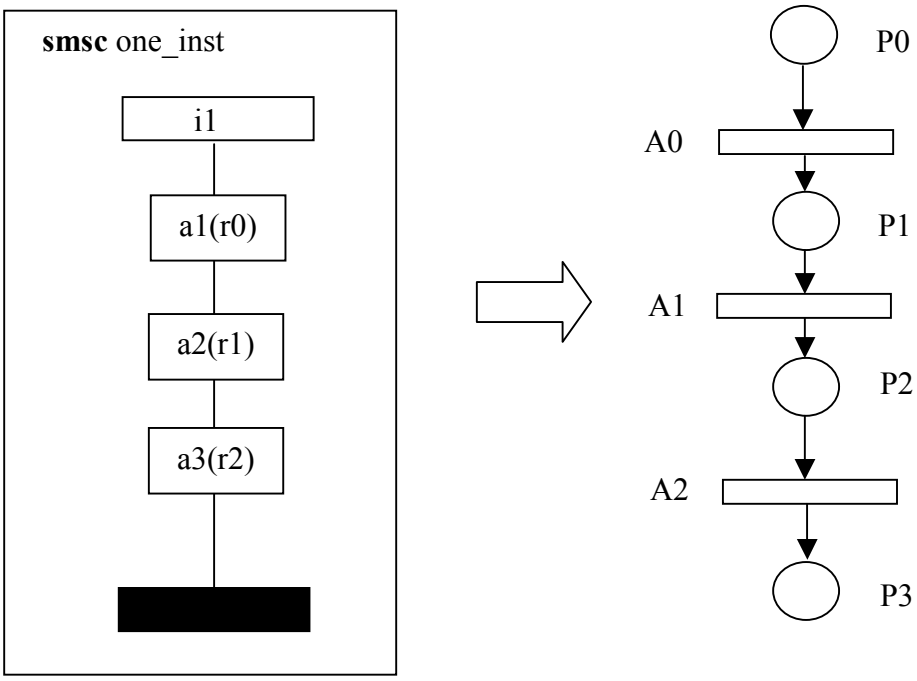

Figure 8 Translating an instance to a SAN

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 8.

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 9.

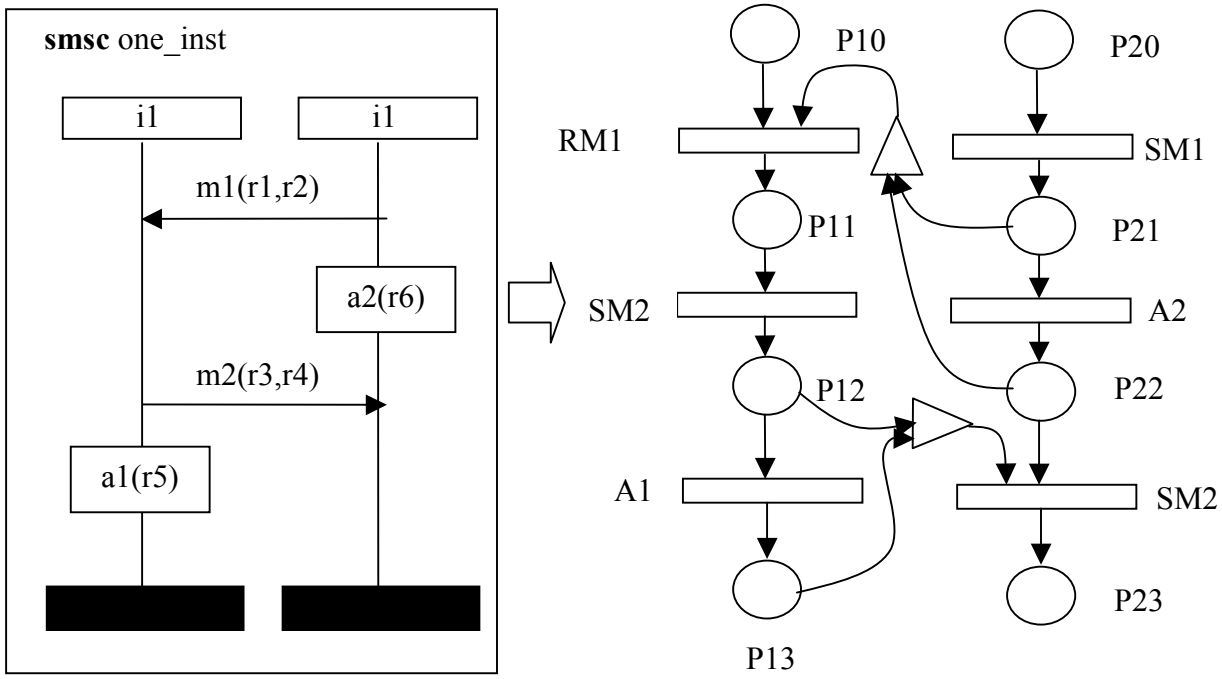

Figure 9 Translation of an SMSC to a SAN

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 9, 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.

3.4.3 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 of the 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 activities. 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 added 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.

3.5 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 FOUR

4. INTEGRATING SMSC INTO 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.

4.1 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.

4.2 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.

4.2.1 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).

4.2.2 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.

4.2.3 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.

4.2.4 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.

4.2.5 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.

4.3 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.

4.3.1 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.

4.3.2 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.

4.3.3 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 10) shows an example of an SMSC and its corresponding

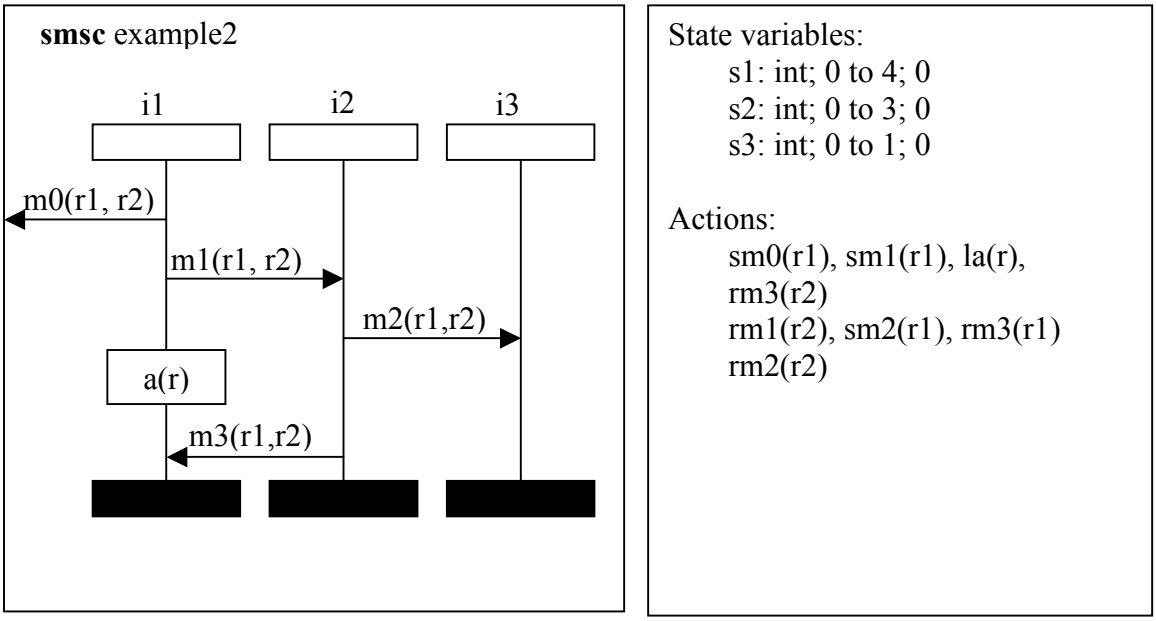

Figure 10 State variables and actions from an SMSC

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 2.

Table 2 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

4.4 Implementing SMSC 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.

4.4.1 Deriving SMSC State Variable Classes

This section describes the implementation of the SMSC state variable classes, including SMSCInst and SMSCCond.

4.4.1.1 The Möbius BaseStateVariableClass

The Möbius BaseStateVariableClass defines methods and data members necessary to implement state variables. Methods defined on BaseStateVariableClass are summarized in Table 3. 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 3 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(BaseActionClass*)	This method appends the specified action to the state variable's object SVAffectingActions
Void appendEnabledAction(BaseActionClass*)	This method appends the specified action to the state variable's SVEnabledActions object
Void copyAffectingActions(List<BaseActionClass>*)	This method copies the data structure passed in and uses it as its list of affecting actions
Void copyEnabledActions(List<BaseActionClass>*)	This method copies the data structure passed in and uses it as its list of enabled actions
Void updateAffects(BaseStateVariableClass*)	This method will notify all the actions on the state variable's SVAffectingActions and 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.

4.4.1.2 The SMSCInst Class

Based on the Möbius BaseStateVariableClass, we derived state variables classes for SMSC models. These state variable classes include SMSCInst, and SMSCCond. The C++ class SMSCInst is defined to represent SMSC instances. The class SMSCCond is to represent SMSC conditions, which are sharable state variables.

The class SMSCInst contains 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 SMSCInst class are shown in Table 4.

Table 4 Data members defined on the SMSCInst class

Data Members	*Descriptions*
short *TheInstValue;	Point to 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 NumCoregions;	The number of coregions defined on this instance;
struct {int start; int end} *Coregions;	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.
List<SMSCActivity> *TheAttachedActivities;	A list of activities in the order as they are given in the SMSC.

In general, the data member **TheInstValue** reflects the current state of the instance by maintaining a pointer to a value that equals to the number of activities that have been executed. The value pointed to by the data member **TheInstValue** increases by 1 after each activity finishes 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 completed. The completion of an activity in a coregion is recorded in a separate Boolean variable array: **CoregionStates**. One must further refer to the **CoregionStates** array to find out whether the number 6 activity has finished.

Another important data member is **TheAttachedActivities**, which maintains a list of activities defined on the instance. The order of the activities, together with the value pointed to by **TheInstValue**, is used to determine whether an activity has finished execution.

Member functions defined on the **SMSCInst** class include **getInstValue(void)**, **setInstValue(short)**, **appendAttachedActivity(SMSCActivity *),** and **CheckFired(AMSCActivty *)**. The member function **getInstValue** is used to examine the current value of this state variable. **setInstValue** is for setting the value of the state variable. The member function **appendAttachedActivity** append an SMSC activity to the list of **TheAttachedActivities.** This function is usually called when initializing the SMSC instance. The member function **CheckFired** is used to check whether a certain activity has finished its execution. **CheckFired** returns TRUE if the specified activity has completed its execution. Otherwise, it returns FALSE.

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.

}

We have implemented a special sharable class: **SMSCCond.** The class **SMSCCond** contains a pointer that points to a short value. A member function specific to the SMSCCond class is **checkCond**, which is used to check whether the condition holds. SMSC conditions are usually used to guard the execution of activities. The activities guarded by an SMSC condition cannot be executed unless the guarding condition is met. Since there is no universal rule to check whether a condition is met, the member function **checkCond** is defined as a virtual function. The default implementation always returns TRUE. Users can derive their own classes from **SMSCCond** and override the default implementation of the **checkCond** with their own implementation.

4.4.2 Deriving SMSC Activity Classes

This section covers the implementation of the SMSC activity class.

4.4.2.1 The Möbius BaseActionClass

BaseActionClass is the implementation of the Möbius entity action. This implementation is quite straightforward. Table 5 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 5 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 6) 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 7.

Table 6 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 7 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	α1, β1
Uniform	UpperBound, LowerBound
Binomial	t, p
NegativeBinomial	s, p
HyperExponential	rate1, rate2, p

Table 8 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 8 shows the complete list of the performance variable related data structures and their corresponding meanings.

4.4.2.2 The SMSCActivity Class

The SMSC Activity class is derived from the Möbius BaseActionClass. 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 activity that precedes it has finished and its guarding conditions are met. 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. Again, the guarding condition must be met if there is one. 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 SMSCActivity.

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

Table 9 Data members defined on the SMSCActivity class

Data Members	*Descriptions*
List<SMSCActivity> *thePrecedingActivities;	Point to a list of activities that must have been executed in order to enable this activity;
List<SMSCCond> *theConditions;	Point to a list of guarding conditions of this activity.
SMSCInst *theInstance;	Point to the instance to which this activity is attached.
SMSCModel *theModel	Point to the model that contains this activity

New member functions are defined to handle these data members. The member function **RegisterPreActivities(SMSCActivity *)** is defined to add the specified activity to the activity list **thePrecedingActivities**. While the function **RegisterConditions** is used to add a guarding condition for the activity. Functions **RegisterInstance** and **RegisterModel** are defined to set the values for data members: **theInstance** and **theModel**, respectively.

The member function **Enabled (void)** is used to check whether the current activity is enabled. It returns a TRUE value if the activity is enabled under the current model state. Otherwise, it returns a FALSE value to indicate that this activity is not enabled. The function **Enabled** is implemented in this way:

- First, it checks whether the current activity has already fired. If not, it continues to the next step. Otherwise, return FALSE.
- Then, check if there are preceding activities defined for this activity. If yes, check if all the preceding activities have finished. If any preceding activity is not finished, return FALSE. Otherwise, continue to the next step.

- Finally, check if there are guarding conditions defined for this activity. If yes, check if all the guarding conditions are met. If any guarding condition is not met, return FALSE. Otherwise, return TRUE.

Note: to check whether an activity has finished its execution, use the member function **checkFired()** defined on the SMSCInst class. That is why an activity should have the data member **theInstance**, from which the activity can know to which instance it is attached.

The member function **Fire( )** is also defined as a pure virtual function as in the BaseActionClass. The reason for not providing an implementation is that the action taken when the activity fires depends on the structure of the model and the system to be modeled. There is no common rule to specify what action should be performed when the activity fires. 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. The only known action is to increase the value of the instance state variable by 1. But this is trivial and can be easily coded into the **Fire** function when implementing the function in the derived class. Since the SMSCActivity class is defined with a pure virtual function, it can only be used as an abstract class, from which one can derive some specific classes for activities in specific models.

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. For example, we can add the activity from which the general ordering originates as the preceding activity to the activity at which the general ordering ends.

4.4.3 Deriving SMSC Model Class

This section defines the implementation of the SMSCModel Class, which is based on the Möbius BaseModelClass.

4.4.3.1 The Möbius BaseModelClass

Models in the Möbius framework act as containers of actions, state variables, and groups. The Möbius BaseModelClass defines methods that are used by solvers or other models to access the Möbius entities in a model. Table 10 shows all the methods defined in the BaseModelClass. These methods can be categorized as list methods, state methods, and composed model methods.

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.

Table 10 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 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.

4.4.3.2 The SMSCModel Class

The SMSCModelClass is derived from the Möbius BaseModelClass. 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 SMSCModel class contains additional data members as shown in Table 11.

Table 11 Data members defined on the SMSCModel class

Data Members	Descriptions
SMSCInst **LocalStateVariables;	All instances in this SMSC model
Short NumConditions;	The number of conditions
SMSCCond **Condtions;	The list of all conditions in this model
Short NumActions;	The number of activities in this model
SMSCActivityClass ** LocalActions;	All activities in a SMSC
Short NumSubModels;	The number of SMSCs in the SubModel list;
SMSCModel **SubModelList;	SMSC Models that follow this model
Double *SubModelWeight;	The probability to select the corresponding sub model.
BOOL Enabled;	The current SMSC is enabled if “true”

The structural information of an SMSC is kept in the SMSCActivity class and SMSCInst class rather than in the SMSCModel class. The SMSCModel class acts as a

container of state variables and activities. In addition, the SMSCModel class also provides methods of composing two or more SMSC models (see section 5.4.4).

The data member **Enabled** defined on the SMSCModel class is a Boolean variable**. 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.

In the SMSCModel class, we provide the implementation of the virtual member functions defined on the BaseModelClass, including **StateSize**, **CurrentState**, **SetState**, **CompareState**, **listActions**, and **listSVs**. These member functions will be used by the Möbius solvers and State Space Generator to communicate with the SMSC models. **StateSize** returns the number of bytes required to record one model state. **CurrentState** copies the model state to a specific memory location, which is passed as the parameter of the **CurrentState** function**. SetState** sets the model to a state based on the data in the memory. The function **CompareState** takes two memory addresses as its parameters. **CompareState** is used to determine if the model states stored in those two locations are actually the same state. It returns TRUE if they are. Otherwise, it should return FALSE. The function **listActions** is used by the Möbius solvers to access all actions defined on the model. **listActions** returns a list that contains pointers to activities of the SMSC model. The function **listSVs** returns a list of pointers to the state variables that have a specific name. All these functions are defined for the AFI and facilitate the information exchange between models and solvers.

4.4.4 Model Composition

The default methods of combining two models by joining the shared state variable in the Möbius framework do not work when specifying the composition of two SMSCs. The

reason is that the state variable defined based on an SMSC instance is not sharable. Therefore, it cannot be used in the Möbius joining operations. SMSCs can be joined vertically, horizontally, or alternatively. This is beyond what can be expressed by the Möbius joining operations. However, the Möbius joining operation can be used to join 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. The SMSCModel class can be used to specify these compositions. First, the vertical composition is achieved by setting **NumSubModels** to 1 and **SubModelList** to point to the succeeding SMSC. 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 in the list: **SubModelList**.

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 **NumSubModels**. The variable **SubModelList** is a pointer array in which each pointer points to an SMSC that follows the current SMSC. **SubModelWeight** contains the probabilities assigned to each subsequent SMSC that are used 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 instances between two or more SMSCs into coregions. This requires significant changes to the structure of the current SMSC. If horizontal composition is specified, it must be resolved before using an SMSCModel object to represent it.

4.5 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.

4.5.1 Analytical Solvers v.s. Simulators

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 utility State Space Generator can be used 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.

4.5.2 State Space Generation Algorithm

The Möbius State Space Generator consists of several libraries, which contain precompiled functions. These functions are linked with user-defined models, such as SMSC models, to generate an executable model. The executable model can generate the model state space. The State Space Generator only uses the Möbius AFI to interact with the model. It has no knowledge of the type of model and is not required to know such information. The algorithm used to generate state space can be described as follows.

Initially, the set that will contain all states is empty. The first action that the State Space Generator takes it to call the member function **StateSize**, which is defined on the model class, to determine the size of memory needed to store one model state. Then, it calls the **CurrentState** AFI function to get the initial state of the model. The initial state is then saved in the set for all states. From the initial state, the State Space Generator will determine the subsequent states that can be reached from the initial state. Before doing that, the State Space Generator calls the **listAction** function to get all the actions defined on this model. For each action, its **Enabled** is called to check if it is enabled. If it is, then the **Fire** function is called. The firing of an action changes the model state. The State Space Generator calls the **CurrentState** function to get the possible changed state. It then checks to see if the state is already in the state set. If not, add the current state to the state set. Otherwise, discard it. The State Space State Generator then uses the **SetState** function to set the model state to the one before firing the action. After all the enabled actions have been fired, the state set may be added with new states that can be reached from the initial state. The State Space Generator continues to check each newly added state to see if newer states can be generated. When no new state can be generated, the State Space Generator stops and returns the state set that contains all model states.

Once the state space is generated, various analytical solvers can be applied to solve the model for the desired performance measures. The analytical solvers only deal with the generated state space. They do not need to interact with the original model, from which the state space is generated. The state transition and the reward calculation are recorded in the data structure that represents the state space.

4.5.3 Model Complexity v.s. Solving Time

The complexity of an SMSC model depends not only on the number of instances, messages and conditions, but also on the structure of the model. The structure of the model is the way that instances, messages, conditions and other model constructs are organized together to represent a certain system. Naturally, the large number of instances and messages implies the higher complexity of the SMSC model. However, sometimes the mode structure plays a more important role in deciding the model complexity. This is because we use the size of state space to measure the complexity given that the model is to be solved analytically and has finite states.

There are two types of constructs that affect the number of the states of an SMSC model. The first type of constructs can increase the number of states, while the second one can reduce the number of model states. The SMSC construct that belongs to the first type is coregion. A coregion specifies a number of activities that can run in parallel or in any sequential order. Thus, more states will be generated for such an SMSC than the one without coregions. The second type of constructs includes messages and general orderings. Messages and general orderings imposed restrictions on the sequential order in which the activities can take place. This makes it impossible that the execution of activities follows certain orders. The exclusion of those execution orders also means that the SMSC model cannot be in some states. Hence, the number of states is reduced.

For example, Figure 11(a) shows an SMSC with one instance and three local activities. This SMSC has 4 states: the initial state and 3 additional states that represent the complete of activities *a1, a2,* and *a3*, respectively. Since activities *a1, a2,* and *a3* can only happen in the given order, the complete of a later activity implies the complete of

the early activities, i.e., the finish of *a2* means the finish of *a1* and the finish of *a3* implies the finish of both *a1* and *a2*. If we add a coregion, which is illustrated in Figure 11(b), to encapsulate these activities, then activities *a1, a2,* and *a3* can execute in parallel. The result SMSC would have 8 states because there is no sequential order and activities can happen in any order. Thus, a coregion increases the number of states.

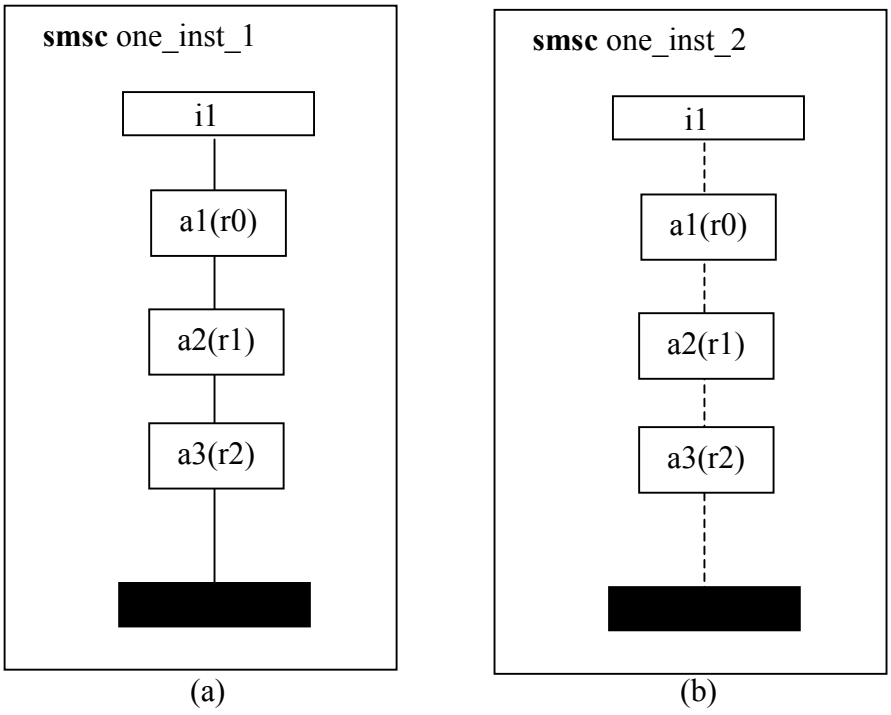

Figure 11 State space without/with coregions

To show that messages or general orderings can reduce the state space size, we first construct an SMSC shown in Figure 12(a). We define two instances and each of them has three local activities. No message is exchanged between the two instances. No general ordering is defined to restrict the execution order between activities on different instances. Although activities of each instance must take place in the specified sequential order, activities between the two instances can actually execute in parallel. The execution of activities on instance *i1* does not affect the execution of those on *i2*. For each instance,

the state variable can take four different values; therefore it has 4 states. Thus, two such instances yield 16 states for the SMSC. Now we define a general ordering between the first activities of both instances *i1* and *i2* (see Figure 12 (b)). This new SMSC shown in

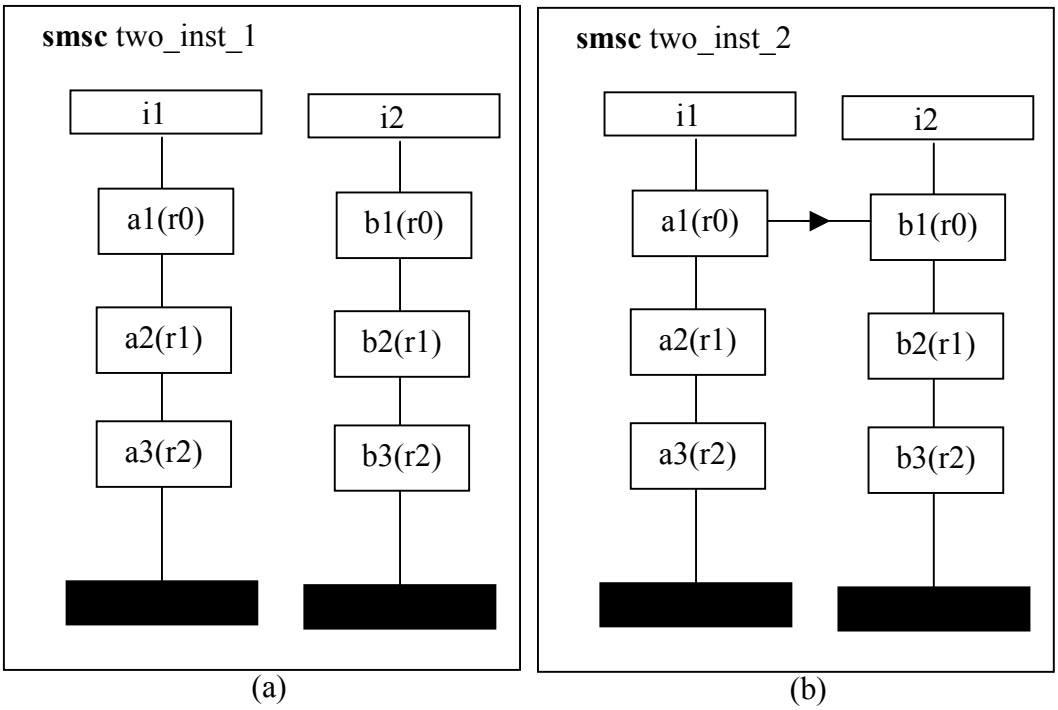

Figure 12 State space without/with general orderings

Figure 12 (b) will have fewer states than the one shown in Figure 12 (a).

The general ordering between activities *a1* and *b1*specifies that activity *b1* can only take place after activity *a1* finishes its execution. This additional restriction on the execution of activities makes it impossible that activities *b1, b2 or b3* be executed before the complete of the activity *a1*. As a result, the number of states of the SMSC shown in Figure 12 (b) is 3-state fewer than that of the SMSC without the general ordering between *a1* and *b1*. Therefore, general orderings that provide additional restrictions can reduce the state space size. Note that general orderings have the same effect in restricting

the execution order as messages. In Figure 12 (b), if we replace the general ordering and the two activities that the general ordering connects with a message that originates from *i1* and ends at *i2,* we have the same result, i.e., the number of states decreases by 3 if compared with the SMSC shown in Figure 12 (a).

If we add even more general orderings to the extend that all activities in the SMSC can

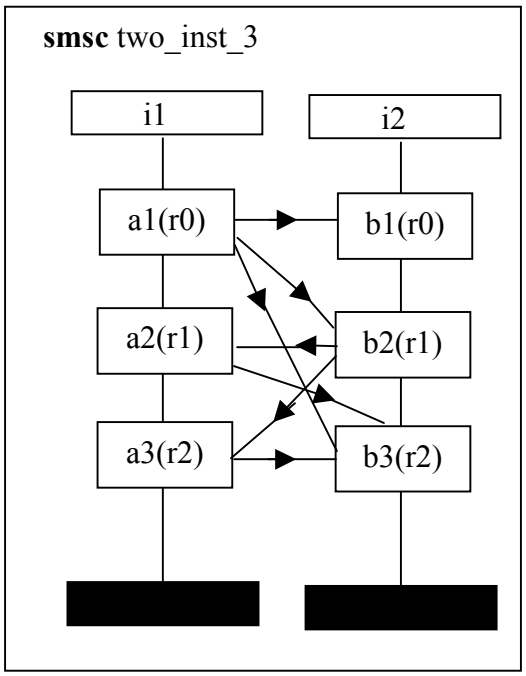

Figure 13 State space with more general orderings

be totally ordered, as is shown in Figure 13, this visually more complicated SMSC actually has the least number of states, which roughly equals to the number of activities. So a visually complicated SMSC does not always mean that it will generate a larger number of states.

In addition to those constructs, the composition of SMSC model also has great impact on the size of its state space. For example, if an SMSC *M1*, which has *S1* number of states, is vertically composed with another SMSC *M2,* which has *S2* number of states,

and the result composed SMSC is called *M3,* the number of states of *M3* is not necessary to be the sum of S1 and S2. Usually, that number is greater than the sum of *S1* and *S2*. Therefore, model composition increases the number of states that the modeled system can take.

The time needed to solve a model is directly related to the state space size of the model. Naturally, the larger the state space, the longer it takes to solve the model. When using the Möbius analytical solvers to solve a model, we need two steps. The first step is to explicitly generate the entire state space using the state space generator. The second step is to choose one analytical solver to solve for the desired reward variables. Hence, the time needed to solve a model can be split into two parts: state space generation time and reward variable solving time. The overall solving time is the sum of those two parts.

To test how efficient the Möbius solvers can handle SMSC models, we use the following example SMSCs (see Figure 14.) The SMSC A has two instances, one guarding condition and one local activity. SMSC B is vertically composed with SMSC A. The SMSC B also has two instances. There are two messages exchanged between these two instances: messages *m1* and *m2*. Another SMSC, SMSC C, is also vertically composed with SMSC A. In other words, SMSC B and C are two alternatives succeeding SMSC A. Inside SMSC C are two setting conditions, one message and one local activity. SMSC C forms a loop, which means that SMSC C is vertically composed with itself.

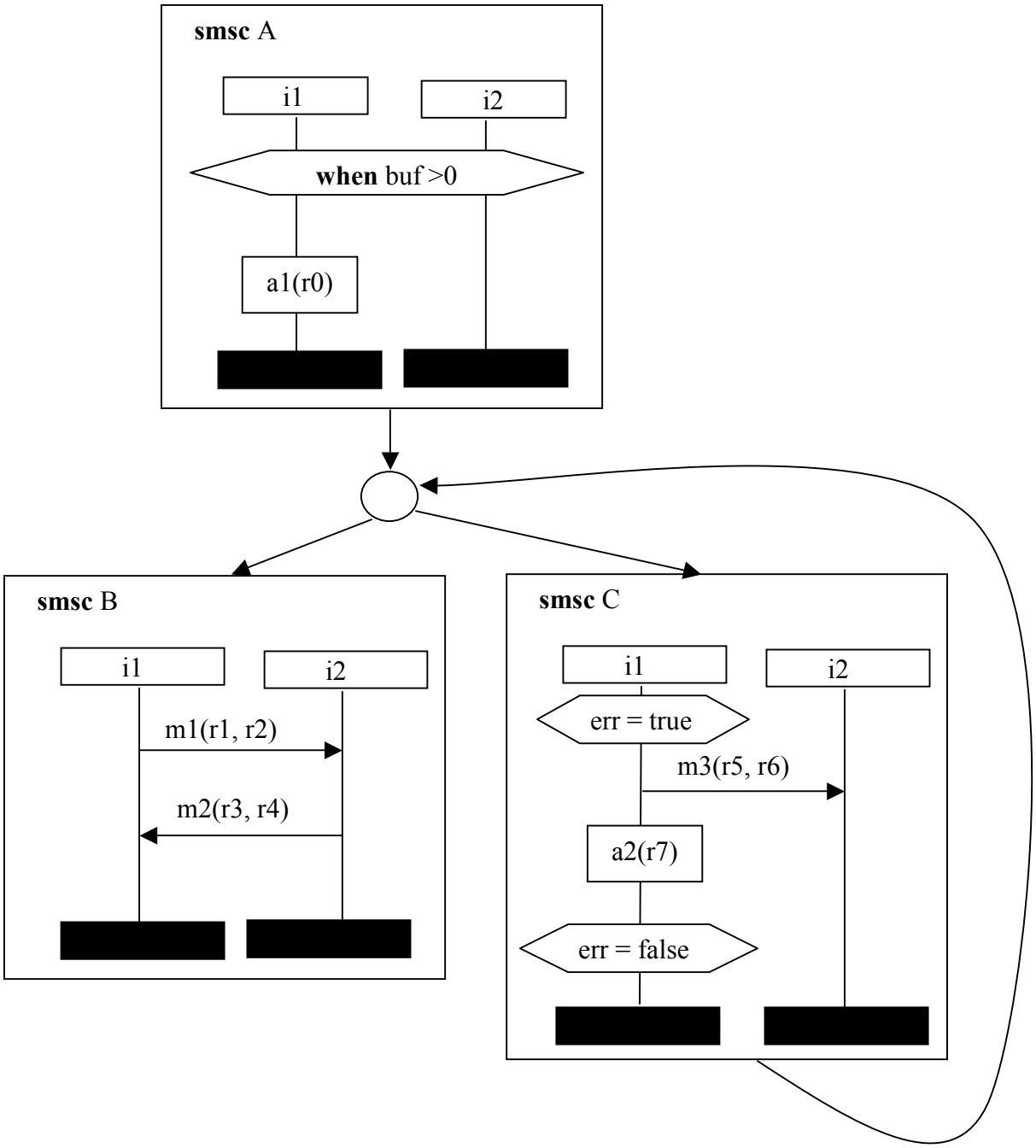

Figure 14 The solved SMSCs

According to the composition describe in Figure 14, SMSC B is also vertically composed with SMSC C. The whole SMSC model consists of 6 instances, two conditions, 3 messages and 4 activities that are not messages. Note we consider setting conditions as activities. Guarding conditions are not activities because they only specify

certain system states. When using the Möbius state space generator to generate the state space, this simple model generates 14 states. This model is so small that the Möbius can solve it instantly.

Based on this simple SMSC model, we build larger models using the Möbius replication/join formalism. The replication /join formalism enables a simple model to be replicated to create a copy of itself. User can specify the number of copies to generate. Usually, this number is defined as a global variable so that different number of copies can be easily specified by assigning different values to the global variable. These copies are joined together through some shared state variables to form a larger model.

In our experiment, the basic model is replicated from 1 to 10 times and it results in 10 different models with increasing complexity. The shared state variable is the one that corresponds to the condition *err*. The number of states generated for each model and the corresponding state space generation time and solving time are shown in Table 12. This experiment was carried out on Windows 2000 machine with 128MB memory and one Intel Pentium III CPU running at 500MHz.

Table 12 Experiment result of model complexity and solving time

Number of Replication	1	2	3	4	5	6	7	8	9	10
States	14	91	455	1820	6188	18564	50388	125970	293930	646646
State space generation time	<1s	<1s	1s	5s	17s	46s	156s	365s	22m	4h30m
Solving time	<1s	1s	10s	70s	416s	27m	1h35m	4h58m	14h46m	52h57m
Total time	<1s	1s	11s	75s	433s	27m46s	1h38m	5h04m	15h06m	57h27m

The number of states with different numbers of replication is shown in Figure 15. We can see that the number of states increases exponentially to the number of replications. One would expect the number of states increase even faster if all the copies of the simple model run in parallel without sharing any state variable. For example, replicating the model 5 times without joining them via a shared state variable would result in a model with 14^5 states, or 537842 states. That number is much larger than the number 6188, which is the number of states from our model. The common state variable shared by those models greatly reduced the number of states of the joined model.

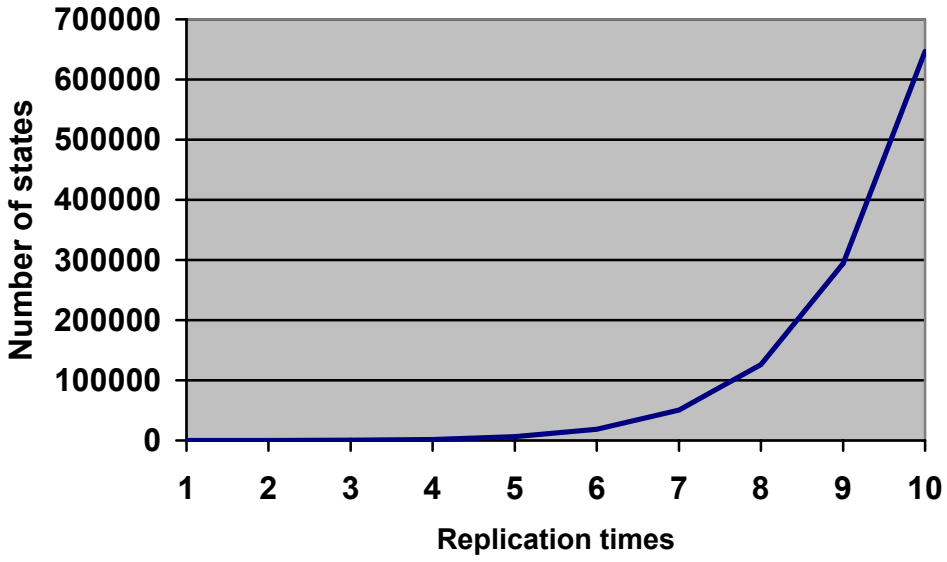

Figure 15 The number of states under different replication times

The state space generation time, reward variable solving time using the Accumulated Reward Solver (ARS) and the total time of solving the models are shown in Figure 16. The time needed to generate the state space is proportional to the number of states. However, when the state space size increases, we noticed a sharp increase in time for the last model, which has 646646 states. This is due to the memory constraint of the machine

used to conduct this experiment. The virtual memory was increased to 400MB and 375MB of it was in use when computing the last model. The available physical memory was less than 1MB at the later stage of computation. The greatly decreased performance must be caused by the disk swap operation in which the operating system consistently swaps data to and from hard disk.

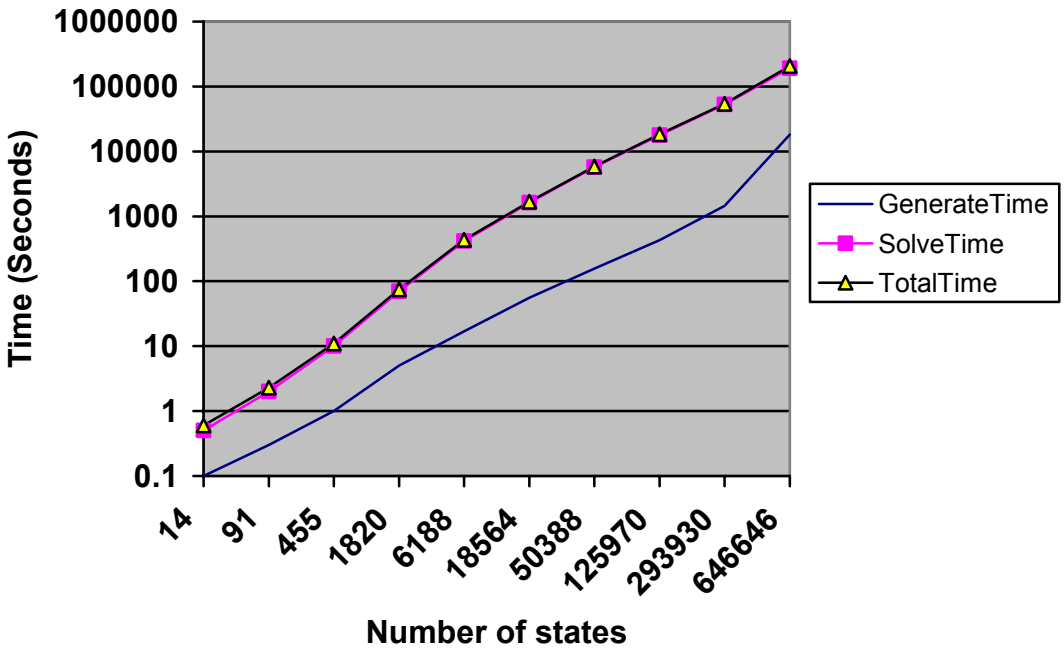

Figure 16 Model solving time

The dominant part of the total model solving time is not the state space generation time but the reward variable solving time. The latter is more than one magnitude higher than the former.

4.6 Summary

In this chapter, we provided a way to define actions and state variables for the SMSC models, and derived the corresponding C++ classes. Some model composition methods

are discussed as well as how they can be easily implemented. We implemented the SMSC formalism and did some experiments to study how efficient the Möbius analytical solvers, especially ARS, can solve SMSC models with different complexities. SMSC is integrated 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 FIVE

5. 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.

5.1 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[36].

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.

5.2 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 17.

The first SMSC shown in Figure 17 (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 17 (b) describes the scenario where an error occurred during the

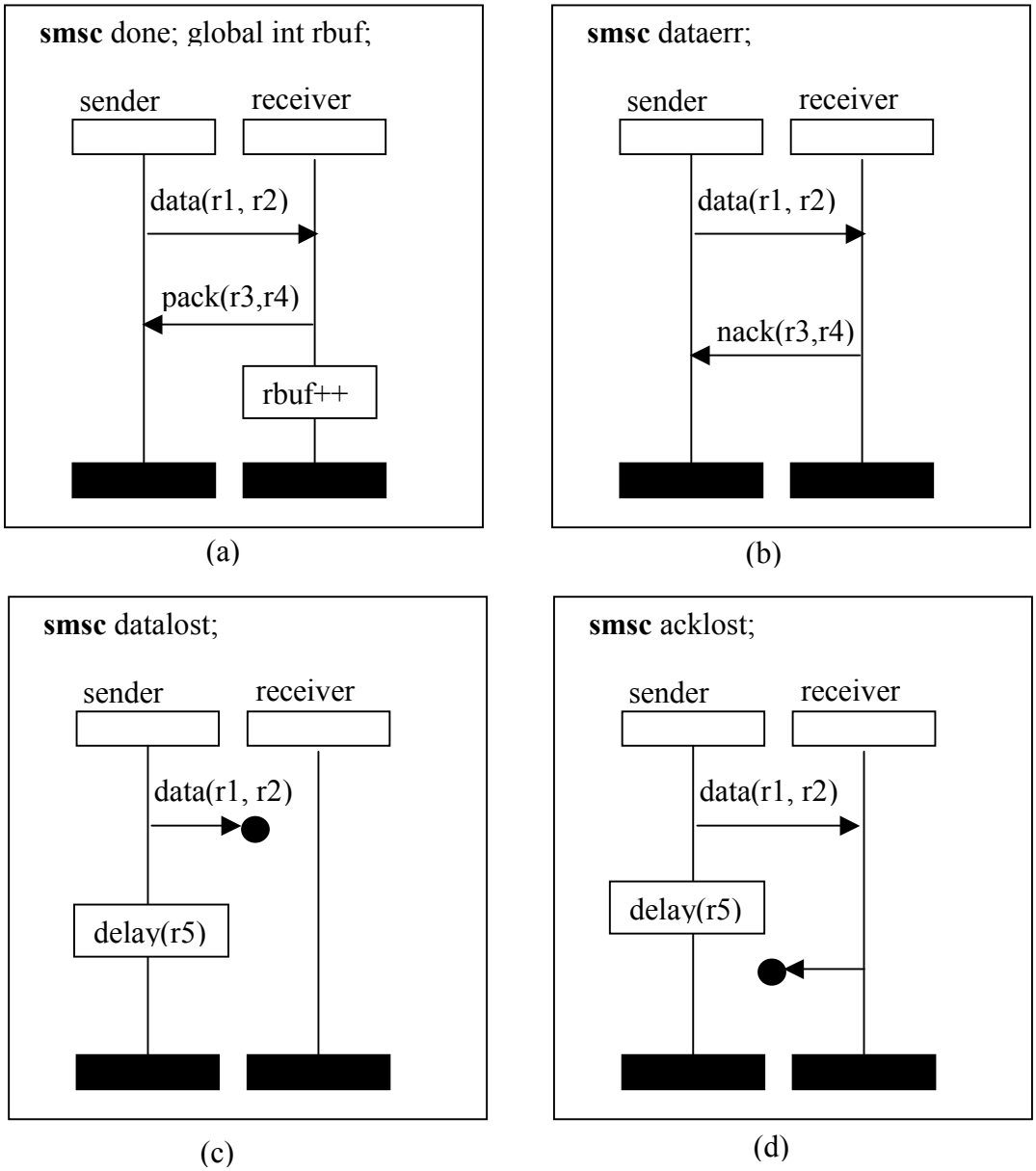

Figure 17 The 4 scenarios of the Stop and Wait protocol

transmission. In this case, a negative acknowledgement is sent back. The scenario shown in Figure 17 (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 17 (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 18 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

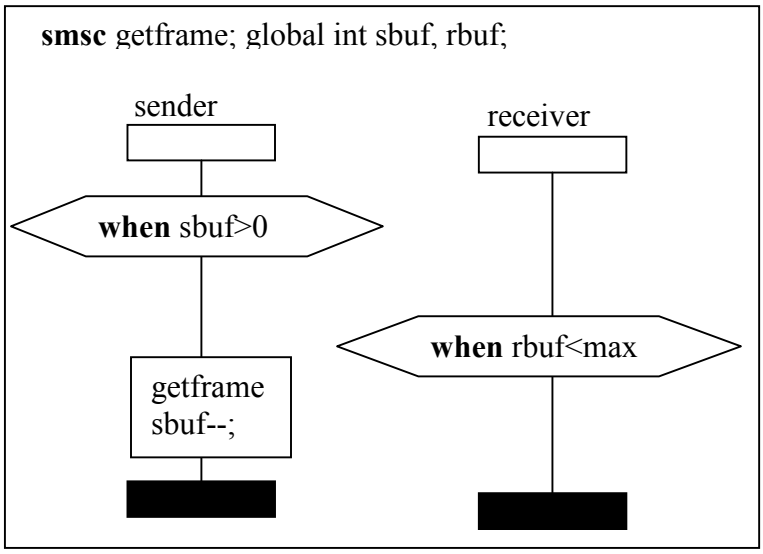

Figure 18 The GetFrame SMSC

wait protocol. The full behavior of this protocol can be described by combining these five SMSCs. Figure 19 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.

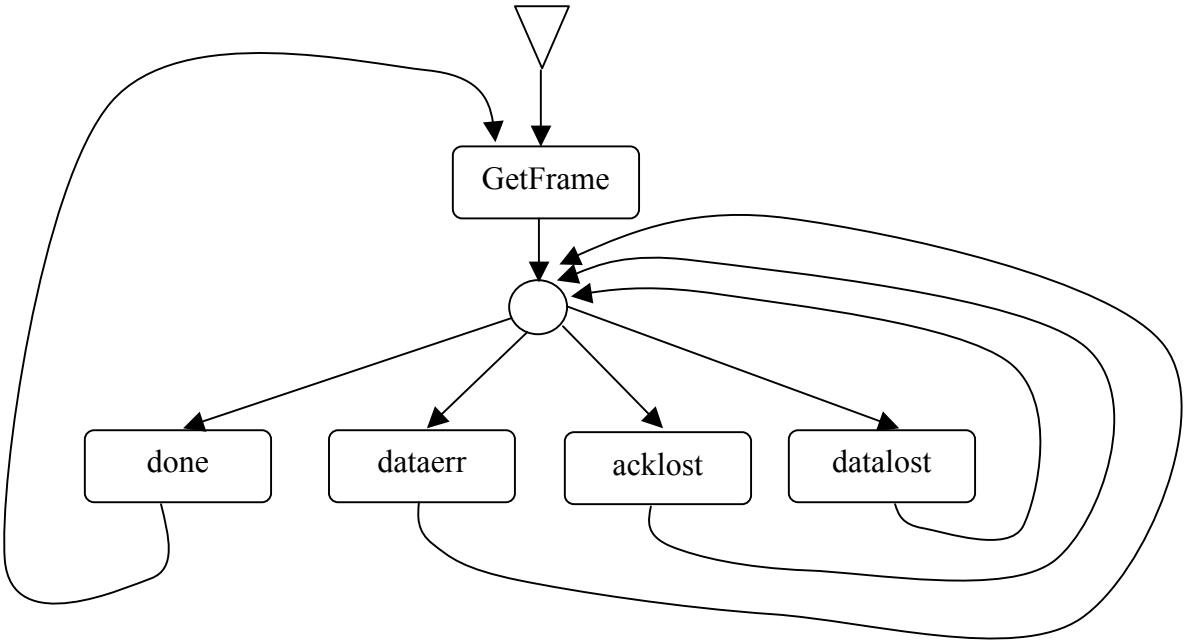

Figure 19 The model of the Stop and Wait protocol

5.3 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 20.

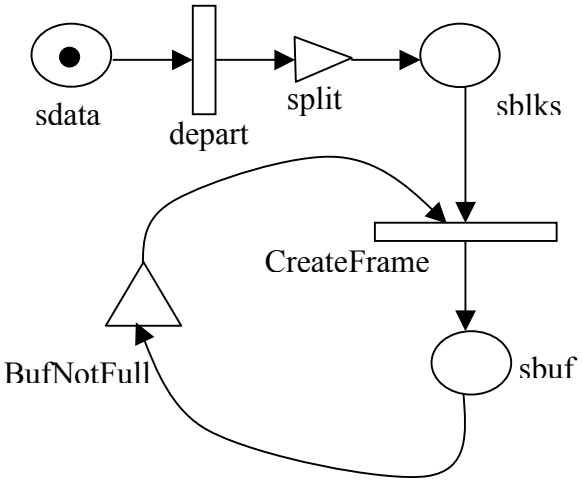

Figure 20 The SAN of the sender

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 21.

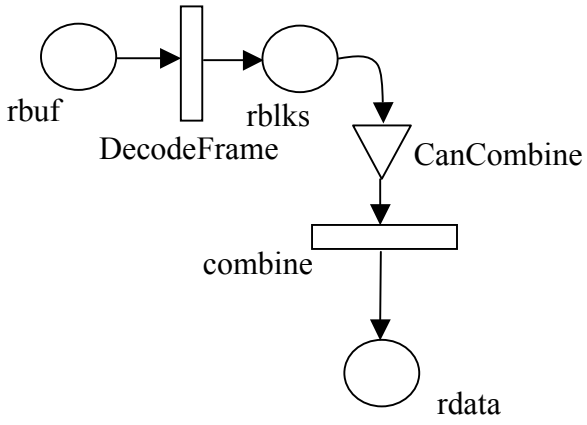

Figure 21 The SAN of the receiver

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.*

5.4 A Heterogeneous Model of the Whole System

The heterogeneous model can be constructed using the Möbius Join and Replicate mechanism as shown in Figure 22.

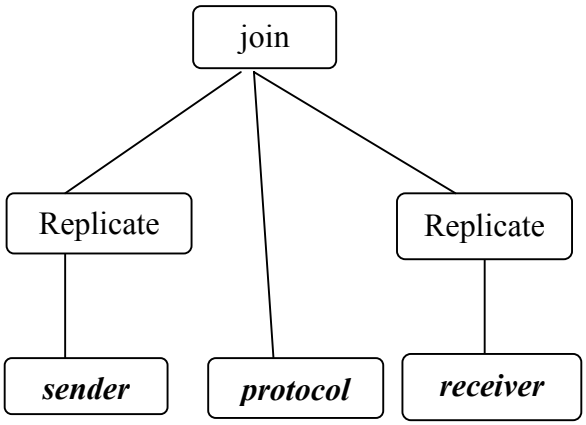

Figure 22 Construct the system model

In Figure 22, ***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.

5.5 Experiment result

To show the Möbius can solve the SMSC model, we defined one reward variable to measure the time that the system spends on handling error data. Whenever error occurs in the channel, the sender would have to retransmit the lost or distorted data frame. The sender may delay for a period of time before it starts to retransmit the data frame if either the data frame or the acknowledgement frame is lost. This period of time is considered as error processing time. We are interested in how the channel error probability and the delay time impact the error processing time.

One additional "condition" *SysInErr* is defined for the SMSC models. Whenever the execution enters one of the SMSCs that describe the error processing scenarios including the SMSC *dataerr*, *acklost*, and *datalost*, the condition *SysInErr* is set to TRUE. The condition *SysInErr* gets reset to FALSE when the execution leaves that SMSC. The reward variable *ErrTime* is a rate reward and will accumulate 1 reward whenever the condition *SysInErr* is TRUE. The channel error probability is defined as a global variable

err_prob. So is the rate associated with the local activity *delay*: *rate_delay*. The Möbius Study Editor can be used to vary the values assigned to those global variables and creates an executable model for each combination of the variable values. In our experiment, we assigned 7 values to *err_prob* (0.01, 0.02, 0.04, 0.08, 0.16, 0.32, 0.64) and 6 values to *rate_delay* (0.125, 0.25, 0.5, 1.0. 2.0. 4.0). Therefore, the Möbius Study Editor generated 42 executable models. The result of this analysis is shown in Figure 23.

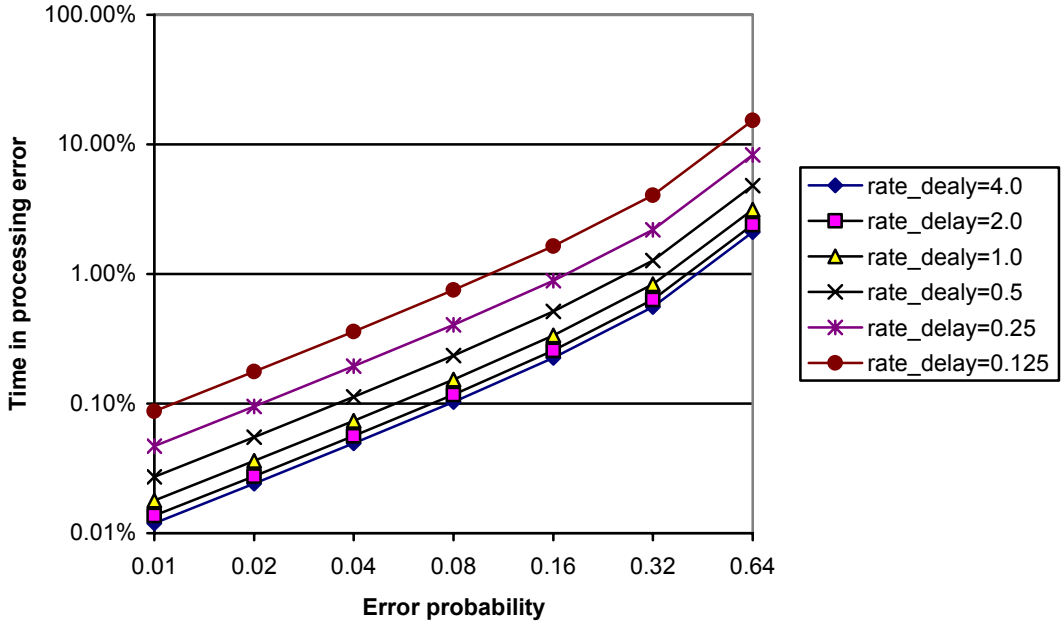

Figure 23 Error processing time of the system

From Figure 23 we can see that the percentage of time in processing error is roughly proportional to the channel error probability. The higher the error probability, the more the time will be spent in processing the error messages. Error processing time is also affected by the delay time. The longer delay time implies that that the sender would have to wait for a longer time before it retransmits data frames. So the longer delay time results in a higher percentage of system time in processing errors. Note rate is defined as

the inverse of time. Therefore, higher rate means shorter delay time. This result seems to imply that we can use 0 delay time to get better performance. But that is not true because the delay time has a lower bound. It must be greater than the round trip time of the messages. Otherwise, unnecessary retransmission for correct data frames will occur and would result in worse performance.

5.6 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 SIX

6. 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 the 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 method of integrating the SMSC formalism into the Möbius framework was investigated. On the basis of this investigation, we discovered that the SMSC formalism could 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 implement the 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 user interface within the Möbius framework. This requires the implementor to collaborate with the Möbius group at University of Illinois at Urbana-Champaign. The user 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 and the model 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 *the 3rd International Workshop 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. D. Daly, D.D. Deavours, A.J. Stillman, P.G. Webster, and W.H. Sanders, *Mobius: An Extensible Tool for Performance and Dependability Modeling.* Lecture notes in computer science, 2000(1786): p. 332-336.

34. J.M. Doyle, *Abstract Model Specification Using the Mobius Modeling Tool*, in *Electrical Engineering*. 2000, University of Illinois at Urbana-Champaign.

35. 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.

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

APPENDIX

A. THE SOURCE CODES OF SMSC CLASSES[3]

1. The SMSC Instance Class

```
/*SMSCInst.h by Zhihe Zhou, May 2002 */
#ifndef SMSCINST_H_
#define SMSCINST_H_
#include "../state/SharableSV.h"
#include "../StandardErrors.h"
#include "../debug.h"
#include "SMSCActivity.h"
#include <iostream.h>
/** This class describes the instance state variable class
for stochastic message sequence charts
 */
 struct Coregion{
     int start;
     int end;
     };

class SMSCInst:public SharableSV<short>{
public:
  /**
    * Default constructor for SMSC instance
    */
  SMSCInst();
  SMSCInst(char* TheInstName);
  SMSCInst(char* TheInstName, short TheInitialValue);
  ~SMSCInst();
  inline short& InstValue(){return *TheInstValue;}
  /** This method returns the instance's value
   */
```

[3] Only partial source codes are listed. The source codes for the detailed implementation of each class are too large to list line by line.

```
  short getInstValue();
  /** This method sets the value of the instance
   *
   *  @param TheNewValue The new value of the SMSC instance
value
   */
  void  setInstValue(short TheNewValue);
  /* This method checks if a specified activitiy is fired */
  bool checkFired( SMSCActivity *act);
  /* add an attached activity */
  void appendAttachedActivity(SMSCActivity *act);
protected:
  /**
    * The protected data structure that holds the instance's
value
    */
  short* TheInstValue;
  /* a list of activities attached to the instance in the
order that
   * they are specified on this instance */
  List<BaseGroupClass> *TheAttachedActivities;
  int NumCoregions;
  bool *CoregionStates;
  struct Coregion *Coregtions;
}; // end class SMSCInst
#endif
```

2. The SMSC Activity Class

```
/* SMSCActivity.h by Zhihe Zhou May 2002. */
#ifndef SMSCACTIVITY_H_
#define SMSCACTIVITY_H_

#include "../BaseGroupClass.h"
#include "../debug.h"
```

```
#include "SMSCCond.h"

class SMSCModel;
class SMSCInst;
/**
 *  This class describes the operation of SMSC Activities --
the
 *  formalism specific action for SMSC models
 */
class SMSCActivity:public BaseGroupClass{
 public:
  bool NewEnabled;
  bool OldEnabled;
  int NumDummyStateVariables;
  BaseStateVariableClass*** DummyList;

  /* These three data members govern the enabling status of
the smsc activity */
  List<BaseGroupClass> *thePrecedingActivities;
  List<BaseStateVariableClass> *theConditions;
  SMSCInst *theInstance; /* An smsc activity must be
attached to an instance */
  SMSCActivity();
  ~SMSCActivity();
  SMSCActivity(char*  TheActionName,
            int TheGroupIDNumber,
            Distribution TheDistributionType,
            ExecutionPolicyType TheExecutionPolicy,
            int TheNumberOfAffectedStateVariables,
            int TheNumberOfEnablingStateVariables,
            int TheNumberOfDummyStateVariables,
            bool TheReactivation);
  void ActivityInitialize(char* TheActionName,
                 int TheGroupIDNumber,
                 Distribution TheDistributionType,
                 ExecutionPolicyType TheExecutionPolicy,
```

```
                int TheNumberOfAffectedStateVariables,
                int TheNumberOfEnablingStateVariables,
                int TheNumberOfDummyStateVariables,
                bool TheReactivation);

  virtual void LinkVariables()=0;
  //virtual BaseActionClass* Fire()=0;
  bool SelectAction();
  void CalculateWeightDistribution();
  bool EnablingChange();
  double Probability(BaseActionClass* TheAction);
  bool IsAnAffectingStateVariable(BaseStateVariableClass*
TheStateVariable);
  bool Enabled();
  double Weight(){
     return 1.0;
  }
  bool ReactivationPredicate()
  {
     return false;
  }
  bool ReactivationFunction(){return false;}
  int Rank(){return 1;}
  virtual BaseActionClass* Fire()=0;

  void RegisterModel(SMSCModel *model);
  void RegisterPreActivities(SMSCActivity *activity);
  void RegisterInst(SMSCInst *inst);
  void RegisterCond(SMSCCond *cond);
  void appendDependActivity(SMSCActivity *act);

 protected:
  double* TheDistributionParameters;
  bool Depend;
  SMSCModel *theModel;
  SMSCActivity *theDependActivity;
```

```
};
#endif
```

3. The SMSC Condition Class

```
/*SMSCCond.h by Zhihe Zhou, May 2002 */
#ifndef SMSCCONDITION_H_
#define SMSCCONDITION_H_
#include "../state/SharableSV.h"
#include "../StandardErrors.h"
#include "../debug.h"
#include <iostream.h>
/** This class describes one sharable variable class for
stochastic message sequence charts
 */
class SMSCCond:public SharableSV<short>{
public:
  /**
    * Default constructor for SMSC instance
    */
  SMSCCond();
  SMSCCond(char* TheCondName);
  SMSCCond(char* TheCondName, short TheInitialValue);
  ~SMSCCond(){delete TheState;};

  inline short& CondValue(){return *TheCondValue;}
  /** This method returns the instance's value (by value)
   */
  short getCondValue();
  /** This method sets the value of the instance
   *
   *  @param TheNewValue The new value of the SMSC instance
value
   */
  void  setCondValue(short TheNewValue);
```

```
  /* check if the condition holds */
  virtual bool checkCond();

protected:
  /**
    * The protected data structure that holds the
condition's value
    */
  short* TheCondValue;
}; // end class SMSCCondition
#endif
```

4. The SMSC Model Class

```
*SMSCModel.h by Zhihe Zhou, May 2002 */
#ifndef SMSCMODEL_H_
#define SMSCMODEL_H_
#include "../BaseModelClass.h"
#include "../StandardErrors.h"
#include "SMSCActivity.h"
#include "SMSCCond.h"
#include "SMSCInst.h"
#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include "../MobiusTypeDefs.h"
class SMSCModel : public BaseModelClass {
public:
 /**
  * Base Constructor for SMSC Models
  */
  SMSCModel();
 /**
  * Overloaded constructor that take an assortment of
parameters
  *
  * @param TheSMSCModelName The name of the SMSC Model
```

```
  * @param TheNumberOfInstances The number of instances
defined within the model
  * @param TheListOfInstances A list of pointers to the
instances defined
  *        within the model
  * @param TheNumberConditions The number of pointers in the
list
  *        of conditions
  * @param TheListOfConditions An array of pointer to all
the
  *        conditions in the model
  * @param TheNumberOfActivities The number of activities
defined
  *         within the model
  * @param TheListOfActivities A list of pointers to
activities defined
  *         within the model
  * @param TheNumberOfGroups The number of action groups
defined
  *         within the model
  * @param TheListOfGroups A list of pointers to action
groups
  */
  SMSCModel(char* TheSMSCModelName,
        int TheNumberOfInstances,
        SMSCInst** TheListOfInstances,
        int TheNumberOfConditions,
        SMSCCond** TheListOfConditions,
        int TheNumberOfActivities,
        SMSCActivity** TheListOfActivities,
        int TheNumberOfGroups,
        BaseGroupClass** TheListOfGroups);

  /**
   *  SMSCModel destructor -- Deallocates memory taken up by
SMSC model
```

```
   *  if compiled with INCLUDE_DELETE preprocessor command.
   */
  virtual ~SMSCModel();

  /**
   * This method performs the initialization of all internal
data structures.
   *
   * @param TheSMSCModelName The name of the SMSC Model
   * @param TheNumberOfInstnaces The number of instances
defined within the model
   * @param TheListOfInstances A list of pointers to the
instances defined
   *         within the model
   * @param TheNumberConditions The number of pointers in
the list
   *        of conditions
   * @param TheListOfConditions An array of pointer to all
the
   *        conditions in the model
   * @param TheNumberOfActivities The number of activities
defined within
   *        the model
   * @param TheListOfActivities A list of pointers to
activities defined
   *        within the model
   * @param TheNumberOfGroups The number of action groups
defined
   *        within the model
   * @param TheListOfGroups A list of pointers to action
groups
   */
  void initializeSMSCModelNow(char* TheSMSCModelName,
          int TheNumberOfInstances,
          SMSCInst** TheListOfInstances,
          int TheNumberOfConditions,
```

```
          SMSCCond** TheListOfConditions,
          int TheNumberOfActivities,
          SMSCActivity** TheListOfActivities,
          int TheNumberOfGroups,
          BaseGroupClass** TheListOfGroups
          );
  /**
   *  This method places the state of the SMSC into the
memory
   *  location specified
   *
   * @param TheStateStorageLocation The location where the
current state of
   *         the  model should be saved.
   */
  void CurrentState(void* TheStateStorageLocation);

  /**
   * This method returns an array of pointers to all the
activities in
   *  the SMSC Model
   *
   *  @return An array of pointers to activities (casted as
BaseActionClass
   *           pointers).
   */
  void  listActions(List<BaseActionClass>* TheContainer);

  /**
   *  This method returns an array of pointers to activities
with the
   *  specified name (not that the array only contains one
or zero elements
   *  if called upon an atomic model).
   *
```

```
   *  @param TheSpecificActionName The name of the specified
activity
   *  @return An array of pointers to activities with the
specified name
   */
  void listActions(char* TheSpecificActionName,
            List<BaseActionClass>* TheContainer);
  /**
   *  This method returns a list of action groups defined
within the SMSC Model
   *
   * @return An array of pointers to action groups defined
within the model
   */
  void listGroups(List<BaseGroupClass>* TheContainer);
  /**
   * This method returns the size of the SMSC model's state
(in bytes).
   *
   *  @return The size of the model state (in bytes)
   */
  int StateSize();

  /**
   * This method set the model state to the one located at
the specified
   *  memory location.
   *
   * @param TheModelStateLocation The location in memory
holding the
   *    state to which the model should be changed.
   */
  void SetState(void* TheModelStateLocation);

  /**
```

```
   * This method compares two SMSC Model states and
determines whether
   * they are the same.
   *
   *  @param StateLocation1 The location of the first SMSC
model state
   *  @param StateLocation2 The location of the second SMSC
model state
   *  @return True if the states are the same, else returns
false
   */
  bool CompareState(void* StateLocation1, void*
StateLocation2);

  /**
   * This method prints infomation about the SMSC to cout
   */
  void PrintSMSCModelInfo();

  /**
   *  This method returns a list of models with the given
name (note this
   *  return trivial for simple atomic models).
   *
   * @param TheSpecifiedModelName The name of the model
   * @return An array of models with the specified name
   */
  void listModels(char* TheSpecifiedModelName,
            List<BaseModelClass>* TheContainer);
  /**
   *  This method returns a list of pointers to SMSC
instances that belong to
   *  the specified model and whose name is the same as the
specified
   *  instance name.
   *
```

```
   *  @param TheSpecifiedModelName The name of the model in
which to search
   *          for the instance
   *  @param TheSpecifiedInstanceName The name of the SMSC
instance
   *  @param TheContainer A data structure containing an
array of
   *         pointers to state variables and the number of
state
   *         variables in the arrary
   */
  void  listSVs(char* TheSpecifiedModelName,
          char* TheSpecifiedInstanceName,
          List<BaseStateVariableClass>* TheSVContainer,
          List<BaseModelClass>* TheModelContainer);
  /**
   *  This method (defined as pure virtual in
BaseModelClass) prints the
   *  state of the SMSC model in a readable format to cout.
   */
  void printState();
  /**
   *  This method (used in RegressionTest)
   *  tests to see if the LocalStateVariables array
   *  has been initialized correctly.
   *
   *  @param TheObject The SMSCModel on which the friend
   *         function operates
   *  @param TheCheckArray The array to be checked
   *         against the LocalStateVariables array
   *  @return True if all the members match, else
   *          false
   */
  friend bool checkLocalSVs(SMSCModel* TheObject,
                  SMSCInst** TheCheckArray);
  /**
```

```
   *  This method (used in RegressionTest)
   *  tests to see if the condition array
   *  has been initialized correctly.
   *
   *  @param TheObject The SMSCModel on which the friend
   *         function operates
   *  @param TheCheckArray The array to be checked
   *         against the condition array
   *  @return True if all the members match, else
   *          false
   */
  friend bool checkConditions(SMSCModel* TheObject,
                      SMSCCond** TheCheckArray);
  /**
   *  This method (used in RegressionTest)
   *  tests to see if the LocalActions array
   *  has been initialized correctly.
   *
   *  @param TheObject The SMSCModel on which the friend
   *         function operates
   *  @param TheCheckArray The array to be checked
   *         against the LocalActionss array
   *  @return True if all the members match, else
   *          false
   */
  friend bool checkLocalActions(SMSCModel* TheObject,
                    SMSCActivity** TheCheckArray);

  /* This variable shows if the model is active, only active
model can
   * have enabled activities *
   */
   bool Enabled;

   /* This variable is used to break the possible infinite
loops when
```

```
    * recursively check submodels
    */
   bool Marked;

   /* These variables are defined to facilitate
vertical/alternative/loop
    * compositon of SMSC models
    */
   int NumSubModels;
   SMSCModel **SubModelList;
   /* when all activities of this model have completed,
    * one of the submodels may be set to active according
    * to their weights
    */
   double *SubModelWeight;

 void SetSubModels(int TheNumOfSubModels, SMSCModel**
TheListOfSubModels, double* TheWeightOfSubModels);


 /* activate the model by setting Enabled = true, and reset
all instance
  * state variables to zero
  */
 void Activate();

protected:
  /**
   * An array of pointers to instances defined within the
SMSC model
   */
  SMSCInst** LocalStateVariables;
  /**
   *  An array of pointers to all the conditions
   *  in the SMSC model.  The value of these instances
   *  are not considered in the SetState, CurrentState,
```

```
   *  or StateSize method since their value is a function
   *  of some other part of state.
   */
  SMSCCond** Conditions;

  /**
   *  This is the number of conditions in the
   *  SMSC Model.
   */
  int NumConditions;

  /**
   * An array of pointer to activities defined within the
SMSC Model
   */
  SMSCActivity** LocalActions;

  /**
   *     This method is used in the Initialize method to
create all the
   *   appropriate data structures for the SMSC instances.
   *
   *    @param TheListOfInstances An array of pointers to
SMSC Insances
   */
  void SetInstances(SMSCInst** TheListOfInstances);

  /**
   *     This method is used in the Initialize method to
create all the
   *   appropriate data structures for the SMSC conditions.
   *
   *    @param TheListOfConditions An array of pointers to
SMSC
   *          conditions
   */
```

```
  void SetConditions(SMSCCond** TheListOfConditions);


  /**
   *  This method is used in the Initialize method to create
all the
   *  appropriate data structures  for the SMSC activities.
   *
   *    @param TheListOfActivities An array of pointers to
SMSC activities
   */
  void SetActions(SMSCActivity** TheListOfActivities);

  /**
   *  This method is used in the Initialize method to create
all the
   *   appropriate data structures for the SMSC activity
groups.
   *
   *    @param TheListOfGroups An array of pointers to SMSC
groups
   */
  void SetGroups(BaseGroupClass** TheListOfGroups);
  }; // end SMSCModel class

#endif // end SMSCModel
```