Extending Message Sequence Charts for Performance Analysis in Möbius

Fredrick T. Sheldon and Zhihe Zhou
School of Electrical Engineering and Computer Science
Washing State University
USA
{sheldon, zzhou1}@eecs.wsu.edu

Abstract: Message Sequence Chart (MSC) is a formal language to describe the communication behavior between the components of a system. In this paper, we propose a new version of MSC, Stochastic MSC (SMSC), which is a stochastic extension to the traditional MSC. SMSC is suitable for performance analysis of distributed and mobile systems. Such systems are often modeled by a number of components that exchange information by passing messages. SMSC is integrated into the Möbius framework, which is an extensible multi-formalism modeling framework that facilitates interactions between models from different formalisms. Not only can SMSC models be solved using the Möbius built-in solver, SMSC formalism also provide an atomic formalism to the Möbius users and can be used as building blocks for large hybrid models.

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 [Sheldon, Xie et al. 2002]. 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. Performance evaluation is an important branch of formal analysis of system properties [Molloy 1982; Ciardo, Marie et al. 1990; Couvillion, Freire et al. 1991; Magott 1992; Haverkort and Niemegeers 1996; Rupe and Kuo 2001]. It regards the quality of service a system can provide. However, not all formalisms are suitable for performance evaluation. For example, Petri Net [Murata 1989] and Process Algebra [Baeten 1994] cannot be used for performance evaluation[1] although they are two famous formal languages in system liveness, deadlock free, or other static property analysis.

Message Sequence Chart (MSC) [ITU-T 1998; ITU-T 1999] 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. However, MSC cannot be used for performance evaluation.

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) [Ciardo 1989], in which transitions are

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

associated with stochastic time information. 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) [Hilston and Hermanns 1994], 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 SMSC formalism can be used for performance analysis. Although many research works had been conducted [Cunter, Muscholl et al. 2001; Damm and Harel 2001; Finkbeiner and Kruger 2001] after MSC was proposed, no one so far 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. SMSC is integrated into the Möbius framework [Daly, Deavours et al. 2000]. Since Möbius is a well-defined framework for multi-formalism modeling and several formalisms (SAN: Stochastic Activity Network [Sanders 1995], PEPA: Performance Evaluation Process Algebra [Hillston 1996], etc.) had been successfully built in [Clark and Sanders 2001; Zhou and Sheldon 2001], 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 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).

2. Background

2.1 Message sequence charts

The full specification of the Message Sequence Charts language can be found at [ITU-T 1999]. Here we briefly introduce the MSC formalism and provide some basic concepts that are necessary to understand this 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 charts, each of them 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 considered to be placed in some big 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 basic MSC with its graphical and textual representations.

A Message Sequence Chart is composed of interacting instances. **Instances** are the primary entities in an MSC. In a specific system, 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.

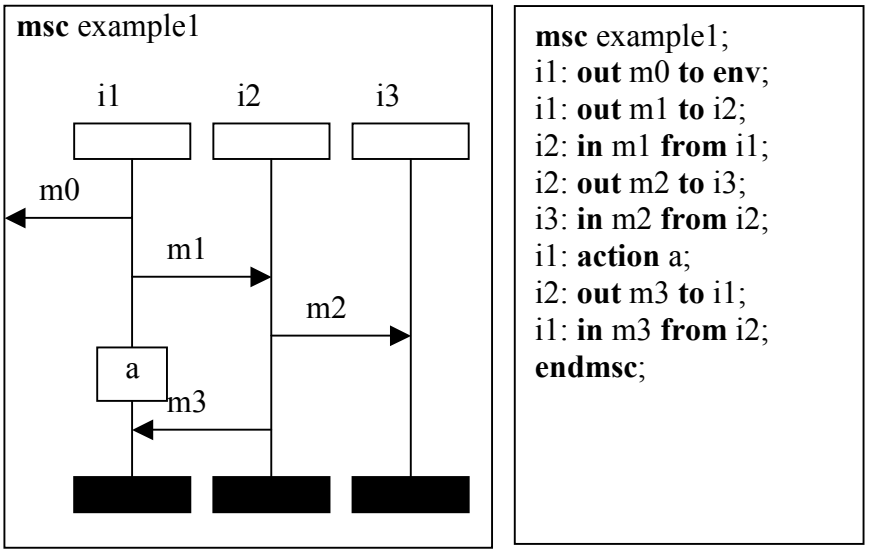


Figure 1. An example of a basic MSC.

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

In addition to messages, **local actions** may be specified in MSCs. A local action describes an internal atomic activity of an instance. It contains either informal text that describes the internal activity, or a formal data statement that defines operations on some data. Graphically, a local action is denoted by an action symbol on an instance with the action string placed in it. The action symbol is a box placed on the instance axis. The part of the axis covered by the action symbol turns invisible or is removed.

An MSC also specifies a partial order, against which the events inside the MSC can take place. There are two basic rules for ordering events. The first rule deals with the ordering of events of the same instance. This rule says that **the events of an instance are executed in the same order as they are given on the vertical axis from top to bottom**. The second rule deals with the order imposed by messages. The key idea for defining this rule is that a message

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

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

The execution of message exchanges and local actions can also be restricted by **condtions**. Conditions are an MSC construct that specifies the system states. Conditions are further classified as **setting conditions** and **guarding conditions**. A setting condition represents the operation that sets the system to certain state. A guarding condition that precedes messages and local actions provides additional restriction the execution of those messages and actions. Only when the condition holds can the messages and actions that following the guarding condition be executed.

The MSC formalism supports structural design. MSCs can be composed vertically or horizontally. Within one MSC, MSC references can be used to refer to other MSCs. Vertical composition means to connect the common instances that share the same name in two MSCs so that the execution of events of the common instances in the succeeding MSC follows the execution of events in the preceding MSC. While in horizontal composition, the events of common instances are interleaved. An MSC can have more than one MSC in vertical composition with it. In this case, the succeeding MSCs are alternatives of each other and such composition is also called alternative composition.

Generally, the way of combining MSCs can be described by a High-level MSC (HMSC), in which MSC references and other constructs are used to specify the composition of MSCs. An HMSC cannot contain instances, messages or local actions although it can use conditions. HMSCs can only use MSC references because the goal of HMSC is to define how the basic MSCs are connected to each other. Figure 2 shows an example of HMSC.

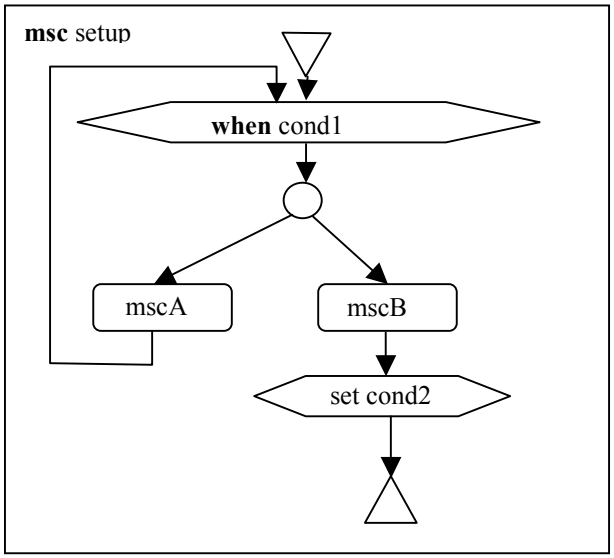


Figure 2. Example of High-level MSC.

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. 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. In other words, 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 have to 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.

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 any models without the knowledge of the underlying formalism. Thus, hybrid models that consist of models from different formalisms can be easily solved.

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 to the Möbius entities for certain formalism. In the implementation of the Möbius tool, state variables are not simple variables, instead, they are implemented as an abstract C++ class: BaseStateVariableClass. So are actions and action groups (BaseActionClass and BaseGroupClass, respectively). One additional class BaseModelClass is defined as the container of the Möbius entities. Each class contains several virtual methods that are part of the AFI. The virtual methods defined on BaseStateVaribleClass include StateSize, SetState, and CurrentState. These methods are used to determine the size of memory needed to store the value of the state variable, set the value of state variable, and get the value of the state variable. For the BaseActionClass, some important virtual methods are Enabled, SampleDistribution, and Fire. The method Enabled is to test whether the action is enabled. The method SampleDistribution is used to determine the time needed to complete the action when the action is enabled. When the action fires, the methods Fire is called. The BaseGroupClass is derived from the BaseActionClass. One additional virtual method is defined: SelectAction, which is used to select the representative for the group. The BaseModelClass also contains AFI functions. These functions include StateSize, SetState, CurrentState, ListActions, ListGroups and ListSVs. The state-related methods have the same meanings as those defined on the BaseStateVaribleClass except they are applied to the model now, not just a state variable. The list functions are used to list the actions, groups or 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 hierarchical model construction method. 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.

3. Stochastic message sequence charts

3.1 Why Stochastic MSC?

The MSC formalism defined in the ITU (International Telecommunication Union) standard [ITU-T 1999] 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 what object. Instances in an MSC actually represent objects of a real system.

Second, certain properties of system behavior can be specified. More precisely, the possible orderings in which actions and messages can occur are defined. An MSC not only contains entities for specifying system objects and their actions, but also imposes a partial order for the events that the system can engage in. We say that only a partial order is implied because there can be events without a defined execution order. These events can happen in any order without violating any rules defined in the MSC formalism. A total order requires that all events can be ordered, directly or indirectly. This is not the case for MSCs. In a summary, MSCs tell us what the system is, what the system does, and how the system should do it. That is why MSC is a Specification Description Language (SDL).

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

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

required to complete the event. The new language is a stochastic extension to MSC. Thus, we call it Stochastic MSC (SMSC).

3.2 Definition of SMSC

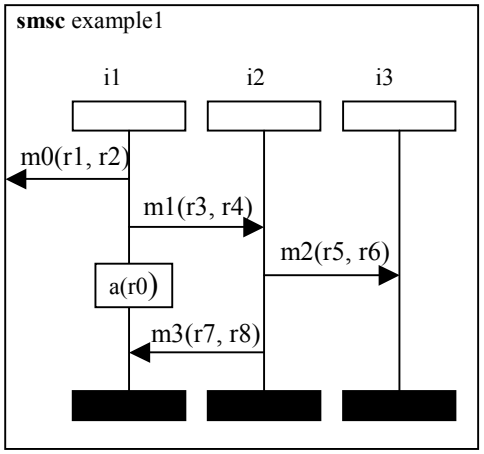
We define SMSC based on the language of MSC:

> **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 is 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 3 shows an example of an SMSC.

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




```
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 3. An SMSC example.

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 3 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.

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

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 3. Note that a new keyword **smsc** is defined to distinguish SMSC from MSC and is used in both the graphical and textural representations.

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 allowed to be non-instantaneous. Therefore, SMSC models provide more information about a system than the MSC model. 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?" The answer to those questions is YES if we carefully define the ordering rules for SMSC.

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 paper, we focus on the exponential distribution only.

SMSC and MSC have the same composition operators. 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. If all activities are associated with zero delay, then the SMSC model is an MSC model.

There are two assumptions made in MSC for precisely ordering events. The assumption of instantaneous events is obvious. If events can last for a period of time, it would be quit possible that another events starts before the 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.

4. Integrating SMSC into the Möbius framework

SMSC language 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, the SMSC formalism is integrated into the Möbius framework. Since the Möbius tool supports multi-formalism modeling, integrating 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 that are available to Möbius so that SMSC can be used for multi-formalism modeling.

4.1 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) Integrate SMSC into the Möbius framework and use the Möbius tool to analyze the SMSC model.

We reject the first method and decide to adopt the second 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, 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 requires that any formalism in the Möbius must implement the AFI and describes its model based on the basic Möbius 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. 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.

Instance state

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:

$$NumInstanc\,eStates = NumInstanc\,eActivites + 1 \tag{4.1}$$

where *NumInstanceStates* is the number of states, and *NumInstacneActivites* denotes the number of activities on the instance. 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 (4.1) 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

$$2^{NumCoregionActivities} \quad (4.2).$$

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 (4.2).

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, integer, or double.

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.

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.

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.

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.

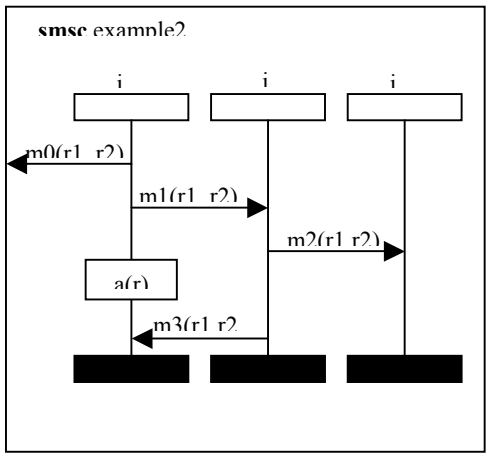
Message Activities

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.

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.



State variables:
s1: int; 0 to 4; 0
s2: int; 0 to 3; 0
s3: int; 0 to 1; 0

Actions:
sm0(r1), sm1(r1), la(r),
rm3(r2)
rm1(r2), sm2(r1), rm3(r1)
rm2(r2)

Figure 4. State variables and actions from an SMSC.

Figure 4 shows an example of an SMSC and its corresponding 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 1.

Table 1. 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.

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 SMSCActivity 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 SMSCModelClass is derived from the Möbius BaseModelClass. The SMSCModelClass is used to organize the state variables and activities for an SMSC model. 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.

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. The SMSC formalism defines its own model composition methods. The SMSCModel class can be used to specify these compositions.

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 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 further 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 5(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 5(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.

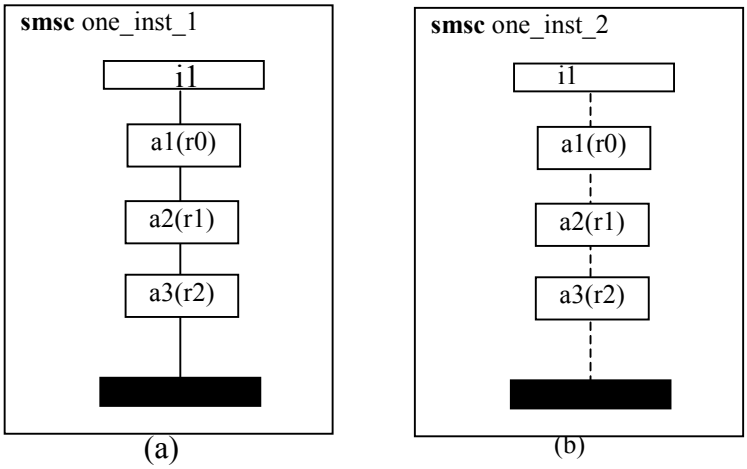


Figure 5. State space without/with coregions.

To show that messages or general orderings can reduce the state space size, we first construct an SMSC shown in Figure 6(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 6 (b)). This new SMSC shown in Figure 6 (b) will have fewer states than the one shown in Figure 6 (a).

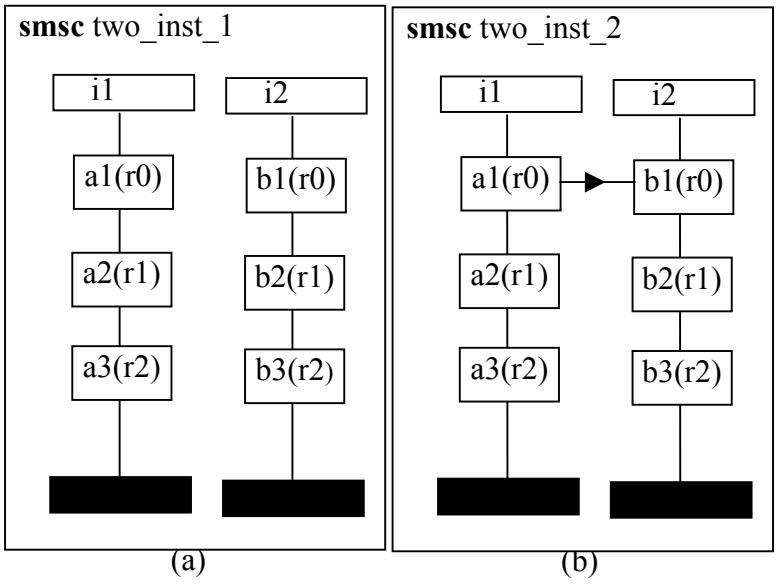


Figure 6. State space without/with general orderings.

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 6 (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 6 (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 6 (a).

If we add even more general orderings to the extend that all activities in the SMSC can be totally ordered, as is shown in Figure 7, 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.

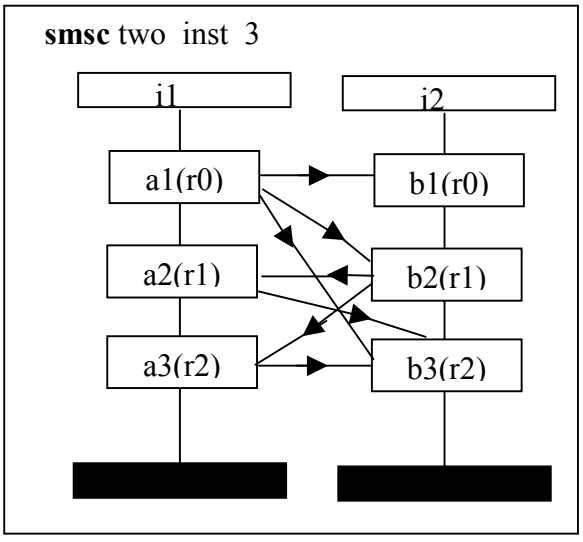


Figure 7. State space with more general orderings.

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 8.) 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.

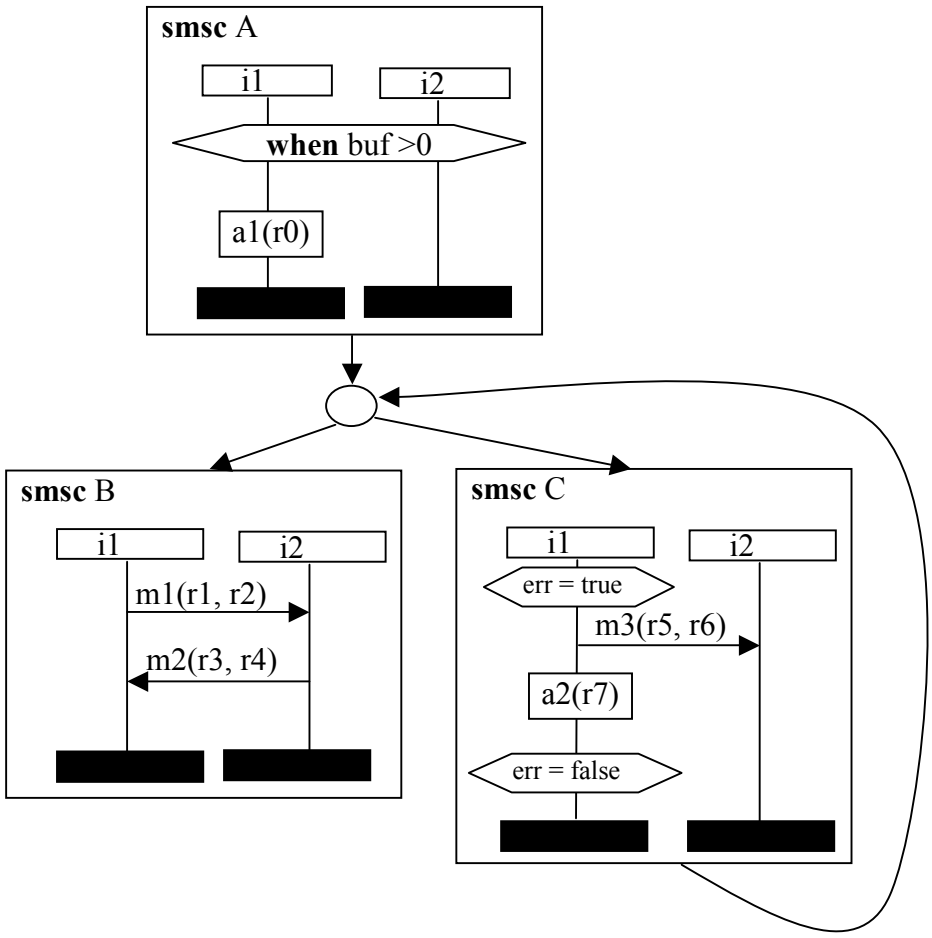


Figure 8. The solved SMSCs.

According to the composition describe in Figure 8, 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 2. This experiment was carried out on Windows 2000 machine with 128MB memory and one Intel Pentium III CPU running at 500MHz.

Table 2. 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

Note: s=second, m=minute, h=hour

The number of states with different numbers of replication is shown in Figure 9. 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.

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 10. 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.

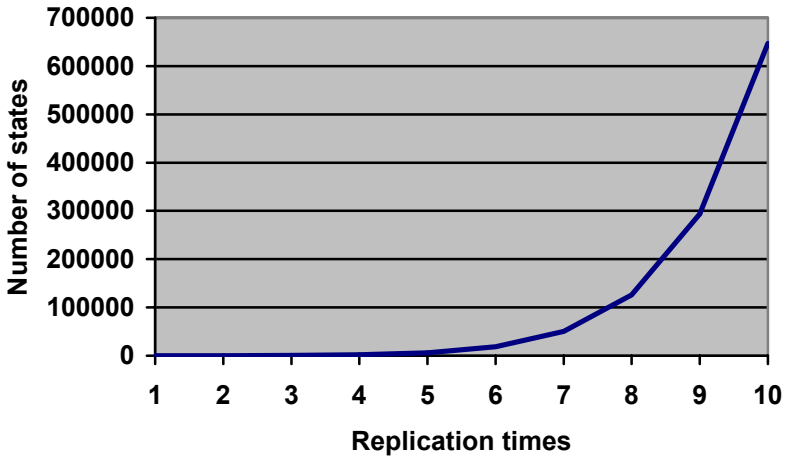


Figure 9. The number of states under different replication times.

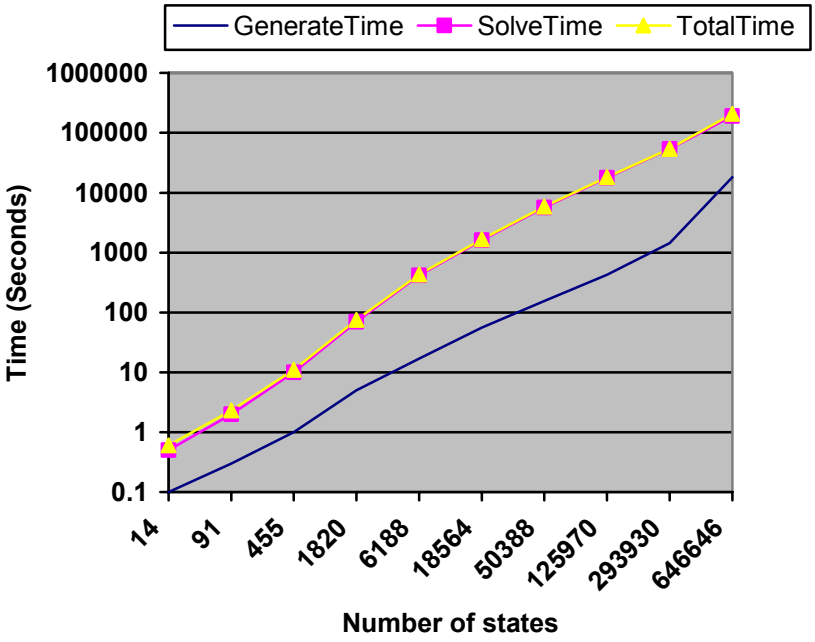


Figure 10. Model solving time with different state space sizes.

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.

5. An example of multi-formalism modeling with SMSC

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 [Tanenbaum 1996].

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 11.

The first SMSC shown in Figure 11 (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 11 (b) describes the scenario where an error occurred during the transmission. In this case, a negative acknowledgement is sent back. The scenario shown in Figure 11 (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 11 (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.

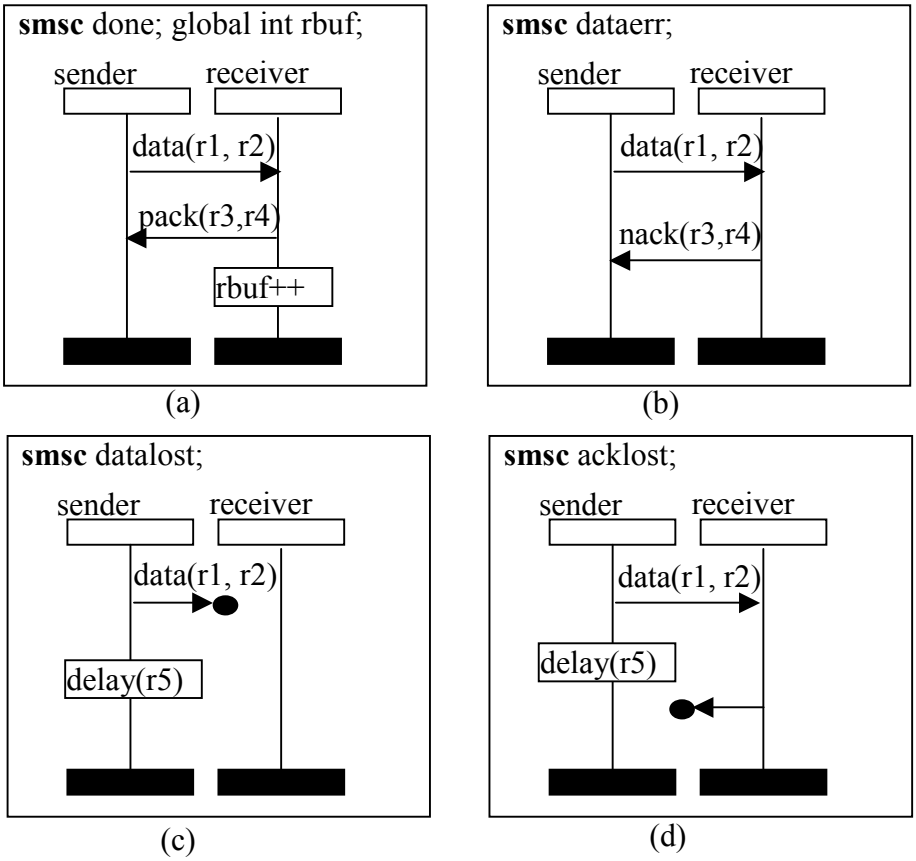


Figure 11. The 4 scenarios of the Stop and Wait protocol.

Figure 12 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 wait protocol. The full behavior of this protocol can be described by combining these five SMSCs. Figure 13 shows the composition methods. The GetFrame SMSC describes the behavior of the sender when it fetching a data frame from the

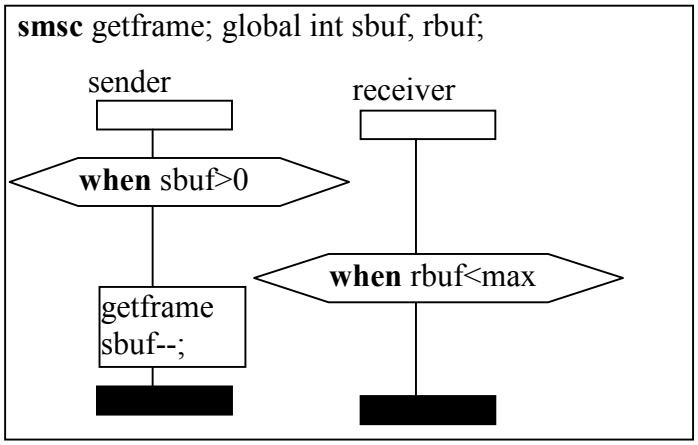


Figure 12. The GetFrame SMSC.

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.

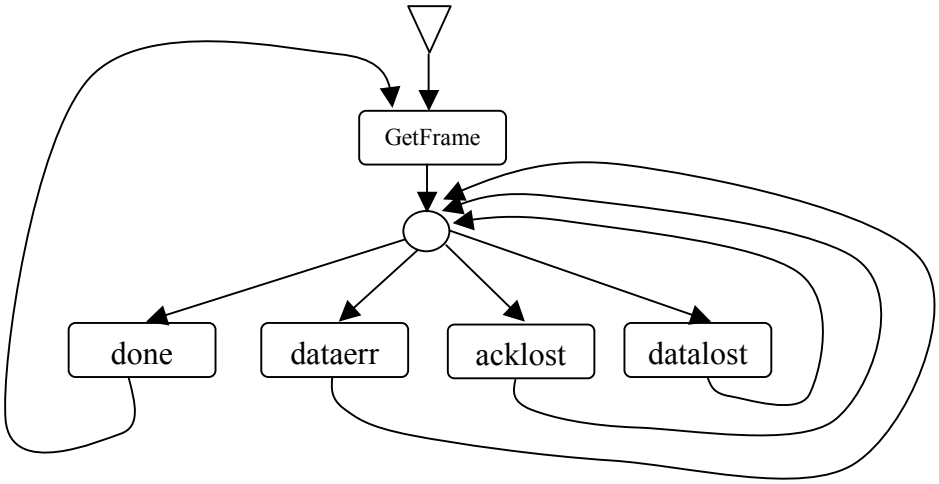


Figure 13. The model of the Stop and Wait protocol.

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 14.

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.

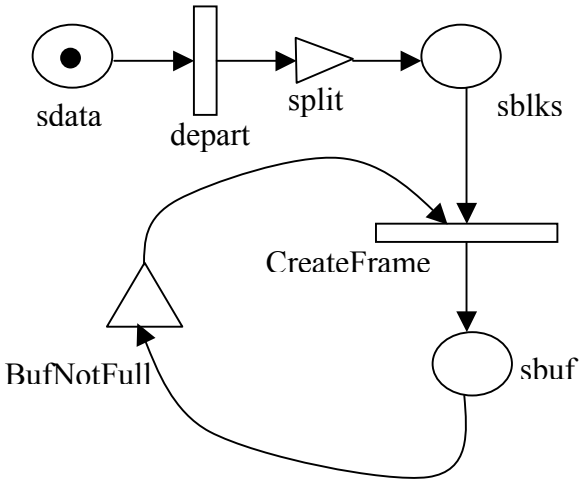


Figure 14. The SAN of the sender.

The SAN model for the data receiving process or the receiver is shown in Figure 15.

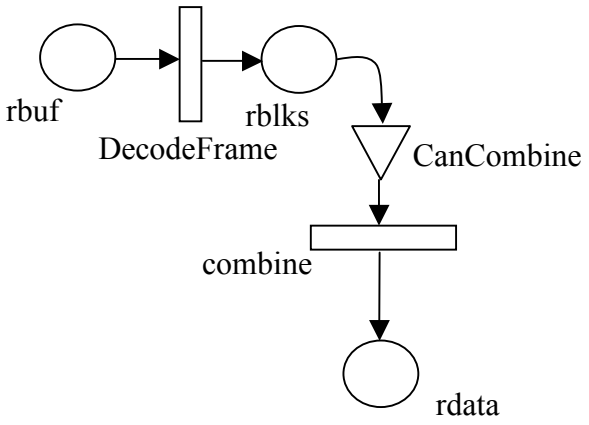


Figure 15. The SAN of the receiver.

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 16.

In Figure 16, ***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.

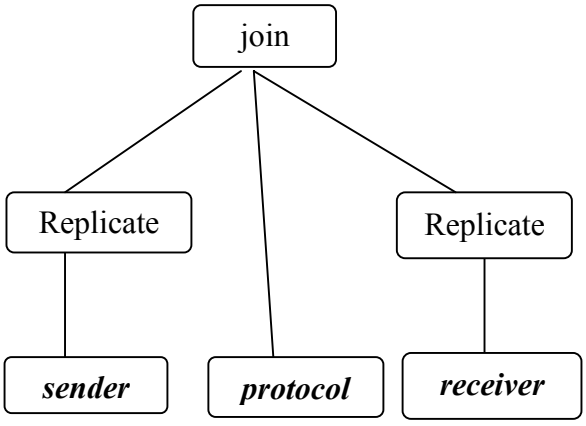


Figure 16. Construct the system model.

Before the models are joined, we must specify the shared state variables. The Join construct in Möbius uses the shared state variable to join different models together, whether they are from the same formalism or different formalisms. In our example, *rbuf* and *sbuf* are shared state variables. In the SAN model, places *rbuf* and *sbuf* are defined as state variables in the Möbius representation. The global data *rbuf* and *sbuf* in the SMSC are also defined as state variables. These state variables are shareable. In fact, they represent the same system components in different models. The number of tokens in the place *sbuf* of the SAN model can be seen by the SMSC model when it checks

its global data s*buf*. The decrement of s*buf* in SMSC model means the removal of a token from the place s*buf* in the SAN model. The increment of the global data *rbuf* in the SMSC model will be interpreted by the SAN model as a token in put into its place *rbuf*. Through these shared state variables, the SAN model and the SMSC model can affect the behavior of each other. The behavior of the whole system is described by models from both formalisms.

6.5 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 17.

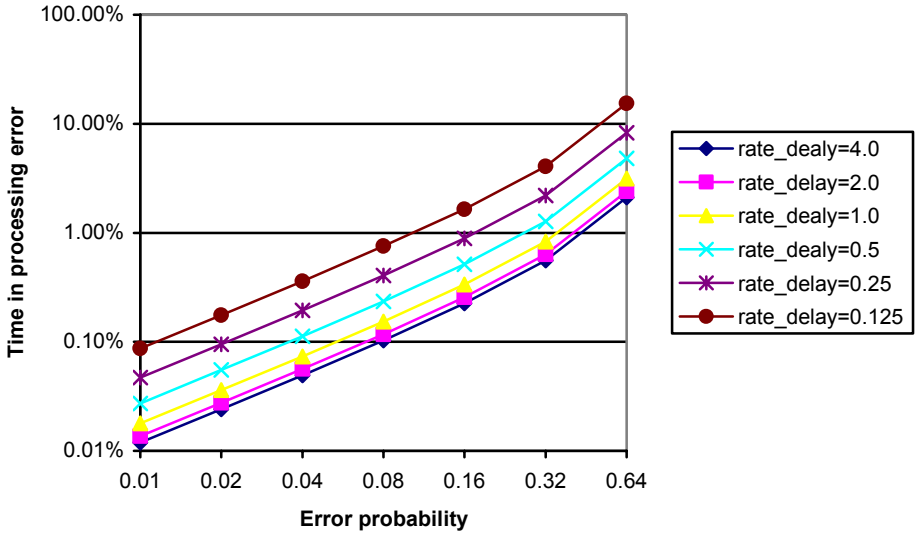


Figure 17. Error processing time of the system.

From Figure 17 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 time in which the system processes errors. Note that rate is defined as the inverse of time. Therefore, higher rate means shorter delay time.

6. Conclusion and future work

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.

7. References

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

Ciardo, G., R. A. Marie, et al. (1990). "Performability Analysis Using Semi-Markov Reward Processes." *IEEE transactions on computers* 39,10: 1251-1264.

Ciardo, G., Muppala J., and Trivedi, K.S. (1989)." SPNP: Stochastic Petri Net Package".In *the 3rd International Workshop on Petri Nets and Performance Models*, Kyoto, Japan, IEEE Computer Society Press, Los Alamitos, CA.

Clark, G. and W. H. Sanders (2001)." 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*, RWTH Aachen, Germany, Berlin: Springer.

Couvillion, J., R. Freire, et al. (1991). "Performability Modeling with UltraSAN." *IEEE software* 8,5: 69-80.

Cunter, E. L., A. Muscholl, et al. (2001). "Compositional Message Sequence Charts." *Lecture Notes in Computer Science*2031: 496-511.

Daly, D., D. D. Deavours, et al. (2000). "Mobius: An Extensible Tool for Performance and Dependability Modeling." *Lecture notes in computer science*1786: 332-336.

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

Finkbeiner, B. and I. Kruger (2001)." Using Message Sequence Charts for Component-based Formal Verification".In *OOPSLA 2001 Workshop on Specification and Verification of Component-based Systems*, Tampa, FL, USA.

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

Hillston, J. (1996). *A Compositional Approach to Performance Modelling*, Cambridge University Press.

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

ITU-T (1998). Formal Semantics of Message Sequence Charts. Geneva.

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

Magott, J. (1992). "Performance evaluation of communicating sequential processes (CSP) using Petri nets." *IEE proceedings. E, Computers and digital techniques.* 139,3: 237-241.

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

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

Rupe, J. and W. Kuo (2001). "Performability of FMS based on stochastic process models." *International journal of production research* 39,Part 1: 139-156.

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

Sheldon, F., G. Xie, et al. (2002). "A Review of Some Rigorous Software Design and Analysis Tools." *Software Focus Journal* 2,4: 140-149.

Tanenbaum, A. S. (1996). *Computer Networks*, Prentice-Hall.

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