Submitted Jan. '03: DSN-2003 Int'l. Performance and Dependability Symposium, San Francisco, CA, June 22nd - 25th, 2003

Integrating Stochastic Message Sequence Charts into Möbius for Performance Analysis

Zhihe Zhou and Frederick T. Sheldon[†]

Software Engineering for Dependable Systems Laboratory©
School of Electrical Engineering and Computer Science
Washington State University
Pullman, Washington 99164-2752, USA
Zzhou@wsu.edu | sheldon@acm.org

Abstract: We propose a formalism called Stochastic Message Sequence Charts (SMSC) and describe how SMSC can be used in the Möbius modeling framework. SMSC is a stochastic extension to the Message Sequence Chart (MSC) formalism used to describe the communication behavior among system components. Compared with MSC, SMSC is suitable for performance analysis. We integrated the SMSC formalism into the framework to enable the use of the Möbius solvers for evaluating the stochastic properties and to leverage the Möbius multiple-formalism modeling feature. This feature enables the SMSC models to interact with models from other formalisms thereby providing valid prediction and/or measurement results. The SMSC formalism provides an atomic formalism for Möbius users and can be used as building blocks for larger hybrid (multi-formalism) models.

Keywords: Message Sequence Charts, Distributed Systems, Stochastic Modeling, Formal Specification, and Performance Analysis.

Word count: 6500.

[†] Sheldon (sheldon@acm.org) is at Oak Ridge National Laboratory (ORNL) and is director of the SEDS (Software Engineering for Dependable Systems) Laboratory.

Integrating Stochastic Message Sequence Charts into Möbius for Performance Analysis

Abstract: We propose a formalism called Stochastic Message Sequence Charts (SMSC) and describe how SMSC can be used in the Möbius modeling framework. SMSC is a stochastic extension to the Message Sequence Chart (MSC) formalism used to describe the communication behavior among system components. Compared with MSC, SMSC *is* suitable for performance analysis. We integrated the SMSC formalism into the framework to enable the use of the Möbius solvers for evaluating the stochastic properties and to leverage the Möbius multiple-formalism modeling feature. This feature enables the SMSC models to interact with models from other formalisms thereby providing valid prediction and/or measurement results. The SMSC formalism provides an atomic formalism for Möbius users and can be used as building blocks for larger hybrid (multi-formalism) models.

Keywords: Message Sequence Charts, Stochastic Modeling, Formal Specification, and Performance Analysis.

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 for developing software and hardware systems, for achieving higher performance and dependability. Performance evaluation is an important branch of formal analysis of system properties [2-7]. It concerns the quality of service a system can provide. However, not all formalisms are suitable for performance evaluation. For example, the original formulations of Petri Nets [8] and Process Algebras [9] cannot be used for performance evaluation and were originally useful for evaluating properties such as system liveness, deadlock free, and other static properties.[1]

Message Sequence Chart (MSC) [10, 11] is a Specification Description Language (SDL) widely used in industry for requirement and design specification as well as test case description. As a formal language, MSC has a well-defined syntax and semantics. MSC models are decomposed into a number of independent message passing instances. System behavior is evaluated through a series of charts indicating interactions between those instances. However, MSC cannot be used for performance evaluation.

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

Consequently, the first problem addressed here is making MSC suitable for performance evaluation. This can be accomplished in a similar fashion as was done for Stochastic Petri Nets (SPNs) and Generalized SPNs (GSPNs) [12, 13], where transitions are associated with stochastic timing information used to evaluate system performance and are widely used for this purpose. A similar extension to PAs exists, known as Stochastic Process Algebra (SPA) [14], where events are associated with random time information, also used for system performance evaluation. Based on the same idea, we have extended MSCs to Stochastic MSC (SMSC) for performance analysis. Although much research has transpired [15-17] since MSC was proposed, it has not been extended to enable the modeling of stochastic properties.

The second problem concerns how to create an analysis tool (i.e., how to solve SMSC models). To address this problem, SMSCs are incorporated into the Möbius framework [18]. Möbius includes a well-defined backplane for multi-formalism modeling that includes several formalisms (SAN: Stochastic Activity Network [19], PEPA: Performance Evaluation Process Algebra [20], etc.), which have been successfully integrated [21, 22]. Therefore SMSC can be integrated into Möbius to enable such models to interact with other built-in Möbius formalisms. By implementing the interfaces required by Möbius, we need not provide analyzers or solvers for the SMSC models. Möbius provides solvers that 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. Message Sequence Charts and the Möbius framework

The full specification of the Message Sequence Charts language can be found at [11]. Here, we briefly introduce the MSC formalism and provide some basic concepts necessary to understand our approach. 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 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. An MSC can be represented graphically or textually. Figure 1 shows an example of a basic MSC with its graphical and textual representations and is composed of the following constructs:

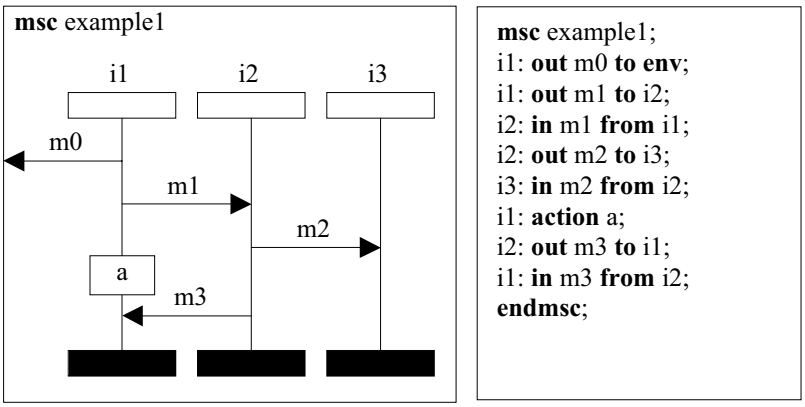


Figure 1. An example of a basic MSC.

- Instances: the primary entities that represent system components.
- Messages: Information exchanged between instances.
- Local Actions: actions happened within one instance without communicating with other instances.
- Conditions: System state that may restrict the occurrence of certain events.
- Coregion: A region where the order of events does not matter.
- General ordering: A construct to explicitly specify the order of two events.
- Reference: refer to another chart.

In addition to the order imposed by coregions and general orderings, an MSC also orders the events using two basic ordering rules:

- *The events of an instance are executed in the same order as they are given on the vertical axis of the chart from top to bottom.*
- *The message-sending event must happen prior to the event of receiving the <u>same</u> message.*

MSC also supports structural design. Generally, the way to combine MSCs is to use a High-level MSC (HMSC), where MSC references and other constructs are used to specify their composition. An HMSC cannot contain instances, messages or local actions although it may employ conditions. HMSCs only use MSC references because the goal of HMSC is to define how the basic MSCs are connected.

Möbius Framework

Möbius 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 [18]. 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. Möbius 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 framework without changing existing tool components.

Möbius defines three basic entities: **state variables**, **actions**, and **action groups** (or **groups**). **State variables** hold the state of the model, or the state of the modeled system. **Actions** are the only entities that can change the values of state variables, thus the state of the model or the system. **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. The hierarchical model construction method is shown in Figure 2.

Möbius defines an Abstract Functional Interface (AFI). The AFI is the core of the 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 AFI consists of functions implemented as C++ virtual methods within the implementation of the C++ classes for Möbius entities. A formalism in the framework must derive its own classes from these basic abstract classes to implement the AFI, i.e., provide their own implementation for those virtual methods.

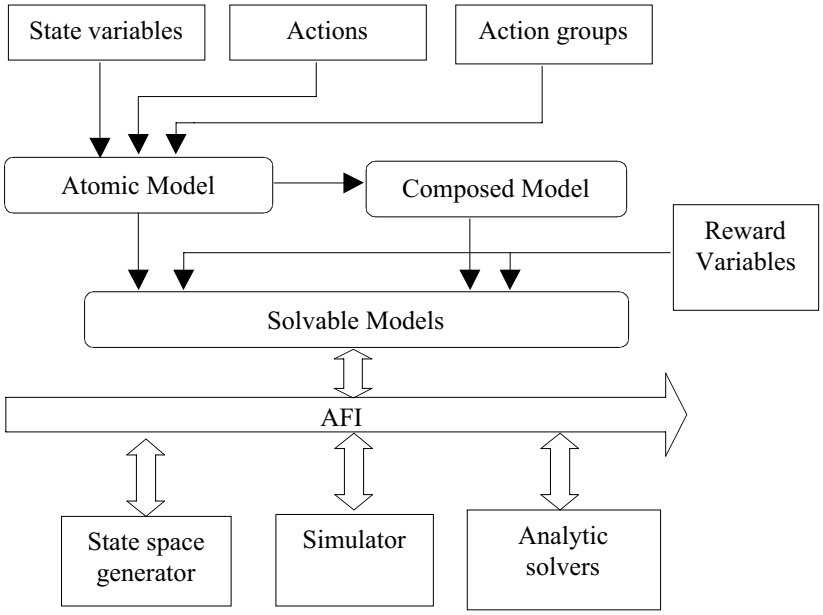


Figure 2 The Möbius framework.

3. Stochastic Message Sequence Charts

In this section, we define SMSC and provide new ordering rules for SMSC. The difference and similarity between SMSC and MSC are explained.

3.1 Definition of SMSC

We define SMSC based on the language of MSC as follows. **An SMSC is an MSC where all events are enhanced to behave as real activities by associating stochastic time information with them. The stochastic time associated with an activity is the time needed to complete the activity.**[2]

The term "Event" is used to describe something that occurs to trigger a set of activities. When an event is associated with time, we call it an "activity." Activity means something that takes time to complete. 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 section (see Figure 3).

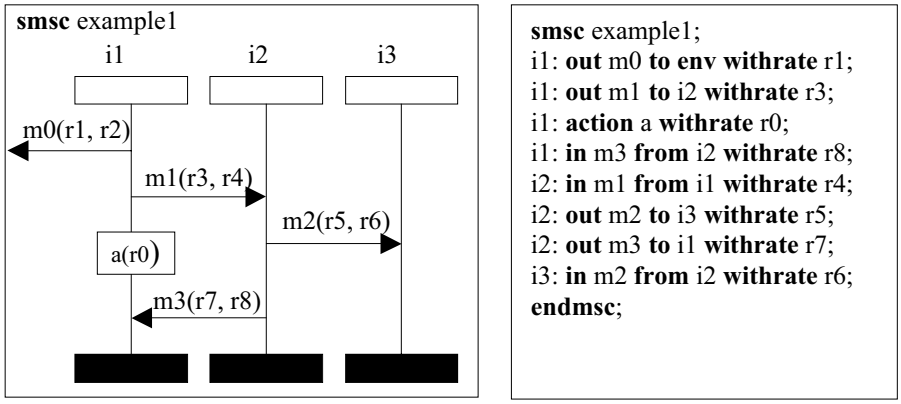


Figure 3. An SMSC example.

The MSC language has two types of events: the events in message passing and the events for local actions. Also, there are two types of activities: message and local action activities or simply local activities. A message in the SMSC language consists of two activities: the activity of sending and the activity of receiving a message. A message is represented the same way as in MSC except the message name is now 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

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

amount of time needed to send the message. *r4* assigns time to the message receiving activity. 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** as shown in Figure 3. Also, note a new keyword **smsc** is defined to distinguish SMSC from MSC and is used in both the graphical and textural representations. Local activities are assigned random time in the same way as messages using one parameter.

3.2 Comparing MSC with SMSC

The SMSC language is different from the MSC because SMSC activities are allowed to be non-instantaneous. Therefore, SMSC models provide more information about a system than the MSC model. However, both of the languages have many similarities.

3.2.1 Constructs

All constructs (instances, messages, local actions, conditions, etc.) defined on MSC are used by SMSC. The graphical representation of a SMSC looks the same as an MSC except for the additional parameters needed to specify time. As for the textual representation, all the keywords defined in MSC are still valid in SMSC. Although new keywords are defined for SMSC, the method and grammar for describing SMSC remains the same.

SMSC and MSC have the same composition operators all of which maintain the same semantics. High-level SMSC (HSMSC) is defined in the same way as HMSC. HSMSC organizes SMSC references using the same nodes defined on HMSC and the organizational interpretation is also the same.

Most new keywords deal with time specification except for the keyword **smsc**, which simply replaces the keyword **msc**. 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. Defining the corresponding keywords and providing the required parameters would enable the specification of other distributions.

3.2.2 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 (i.e., they take no time). If events can last for a period of time, it would be quit possible that another event(s) start before the already started 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 requires that any two events have a specific order. An event either happens before or after the other. Consequently, the execution of events forms a trace that describes system behavior. In SMSC, we relax the first assumption (i.e., events may not be instantaneous). As a result, the second assumption does not hold and is also relaxed. Activities in SMSC can start or finish at the same time. Moreover, this relaxation of the second assumption is more realistic.

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 it's ending, then 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 activity ordering may not be unique for an execution trace of such 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 must be 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 as the same defined for MSC. There are five rules for 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 has finished.*
3) *The activity of sending a message must finish before the activity of receiving the same message can begin.*
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 (*SS*), the event of starting to receive the message (*SR)*, the event of finishing the sending of the message (*FS*), and the event of finishing the receiving of the message (*FR*). The precise restriction for their order is that *SS* must happen before *SR*, while *FR* must happen after *FS*. 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. If we allow the activity that receives a message to start before the completion of the activity that sends the message, we cannot guarantee that the end of receiving the message occurs after the completion of sending the message because both activities are associated with random time.

The fourth and fifth rules are defined for ordering events in a coregion or for being controlled by general orderings. The interpretation is straightforward. Under these rules (i.e., using either the order of starting or of ending events as the order of activities), the order imposed by an SMSC is sure to comply with the partial order imposed by the corresponding MSC if timing information is removed. 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 ordered differently must correspond to the events that have undefined order in the corresponding MSC.

3.2.3 Traces versus 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. The stochastic process enables users to do performance analysis about the system. This is the reason we extend MSC to SMSC.

4. Integrating SMSC into the Möbius Framework

The SMSC language is capable of performance modeling. Since the Möbius tool supports multi-formalism modeling, integrating SMSC into Möbius not only provides a tool for solving SMSC models, but also enables SMSC model to interact with models from other formalisms made available by Möbius.

4.1 Problem Definition

Möbius requires that any formalism in Möbius 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 changes and the ordering of action firings are determined by the structure of the SMSC model. Therefore, before using Möbius to solve an SMSC, the following problems must be addressed:

1) How to define SMSC states and the corresponding state variables, and
2) How to define SMSC actions.

4.2 Identifying State Variables in SMSCs

To define the state of an SMSC, we first examine its components to see what information is necessary to specify state. 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 components contain the information that describes the system state.

Instance state

The state of an instance reflects which activity has been executed. Since an instance imposes a sequential execution order on its activities, it is important to keep the information about the execution of activities to ensure their 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 completed.

Conditions

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 a state change in the model. The type of the state variable for a data member is the same as the type of the data member.

Shareable vs. Non-shareable State Variables

Möbius uses the concept of state sharing to join models from the same or different formalisms. If a state variable is shared with other models then they can also change the value of the state variable. 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 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 completed even though they have not yet been executed. This is referred to 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.

4.3 Identifying Actions in SMSCs

By definition in the Möbius, 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.

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 activity, which receives that same message, performs the receiving activity. Data operations are also 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 changes 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 puts the system in a 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.

Figure 4 shows an example of an SMSC and

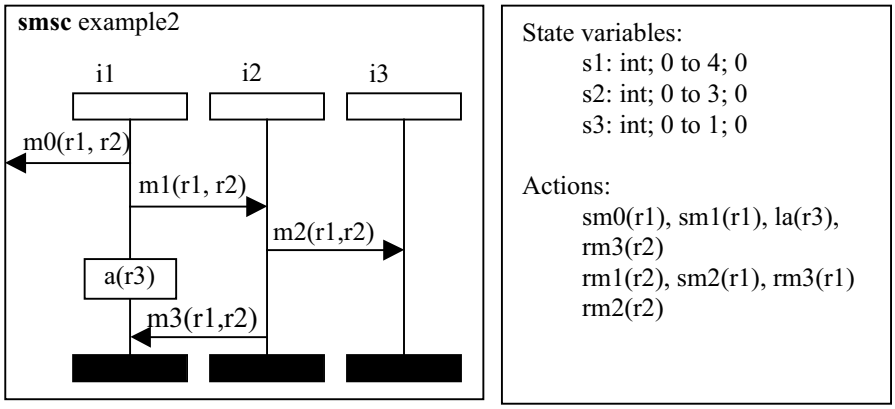


Figure 4. State variables and actions from an SMSC.

its corresponding state variables and actions. Parameters r1, r2 and r3 are the rates associated with activities. 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

4.4 Solving SMSC Models

Once the SMSC models are described using classes derived from the Möbius base classes, we can then solve the models using Möbius built-in solvers to extract estimates or predictions (e.g., reliability, availability, throughput, responsiveness, or other stochastic parameter).

4.4.1 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, which is then used to generate the model state space. The State Space Generator only uses the Möbius AFI to interact with the model (see [23] for details). Once the state space is generated, various analytical solvers are applied to solve the model for the desired performance measures. State transition and reward calculations are recorded in a data structure that represents the SMSC model state space.

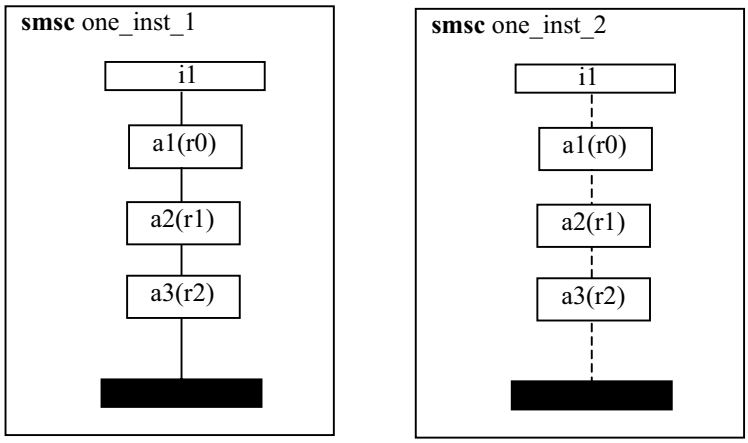
4.4.2 Model Complexity

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, if the SMSC model contains a large number of instances and messages, this implies a higher complexity. However, sometimes the structure of the SMSC model plays a more important role in deciding the model complexity. The state space (size) is used to measure the complexity given that the model is to be solved analytically and has a finite number of states.

There are two types of constructs that affect the number of the states. The first type of construct can increase the number of states, while the second can cause a reduction. The SMSC coregion construct belongs to the first type. A coregion specifies a number of activities that can run in parallel or in any

sequential order and all possible interleavings must be considered (giving rise to many more states). The second type of construct includes messages and general orderings. Messages and general orderings impose restrictions on the sequential order in which the activities can take place, which effectively eliminated certain interleavings resulting in fewer states.

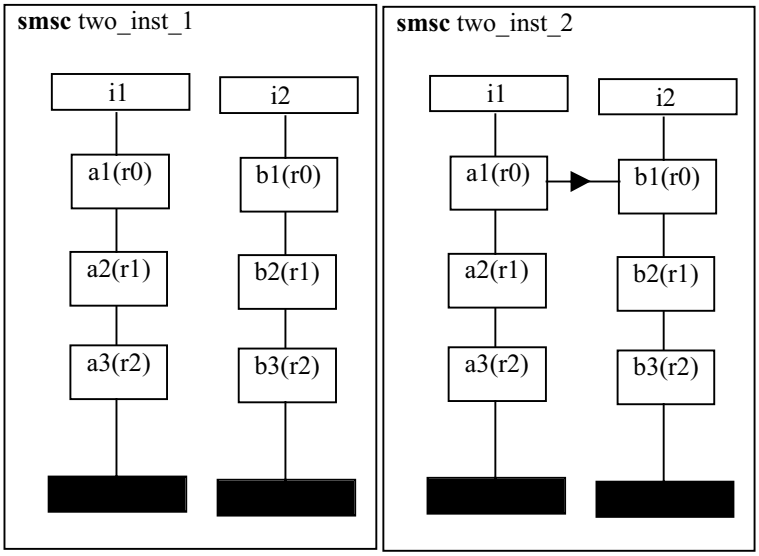
For example, Figure 5(a) shows an SMSC with one instance and three local activities. This SMSC has 4 states: the initial state and 3 additional states that represent the completion of activities *a1, a2,* and *a3*, respectively. Since activities *a1, a2,* and *a3* can only happen in the given order, the completion of a later activity implies the completion of the earlier 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, as illustrated in Figure 5(b), to encapsulate these activities, then activities *a1, a2,* and *a3* can execute in parallel. The resultant SMSC gives 8 states because there is no imposed order and thus, activities happen in any order. Therefore, coregions increase the number of states.



(a) (b)

Figure 5. State space without/with coregions.

To show how messages or general orderings can reduce the state space, we first construct the SMSC shown in Figure 6(a). We define two instances, each of which has three local activities. No message is exchanged between them. 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



(a) (b)

Figure 6. State space without/with general orderings.

sequential order, activities between the two instances can actually execute in parallel. For each instance, the state variable can take four different values; therefore it has 4 states. Thus, two such instances yield 16 states. Now we define a general ordering between the first activities of both instances *i1* and *i2* (see Figure 6 (b)). This general ordering specifies that activity *b1* can only take place after activity *a1* finishes. This additional restriction on the execution of activities makes it impossible for activities *b1, b2 or b3* to occur before the completion of the activity *a1*. Consequently, as shown in Figure 6 (b), the number of states is reduced by 3. Therefore, general orderings that provide additional restrictions can reduce the state space.

In addition to the aforementioned constructs, SMSC model composition has also a great impact on the 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 composed SMSC is called *M3,* the number of states of *M3* is not necessarily the sum of *S1* and *S2*. Usually, that number is greater than the sum of *S1* and *S2* but less than the worst case, the product of *S1* and *S2*. Therefore, model composition increases the number of states that the modeled system can take.

5. Example: Modeling A Communication System

We consider a simple system with two computers connected via cable. The processes running on one computer send files to those running on another. The communication protocol used by the data link layer is the stop and wait protocol [24]. The sending and receiving processes are modeled as Stochastic Activity Networks (SANs)[19]. The stop and wait protocol is modeled using SMSCs.

5.1 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. Until obtaining the receivers' acknowledgement, the sender cannot start sending the next block. This prevents a fast sender from flooding a slow receiver with 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 a retransmission technique must be used. The sender starts a timer after transmitting 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, and if correct, a positive acknowledgement is sent back. Otherwise, a negative acknowledgement is sent back. 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 describes a scenario for their behavior using this protocol (see Figure 7).

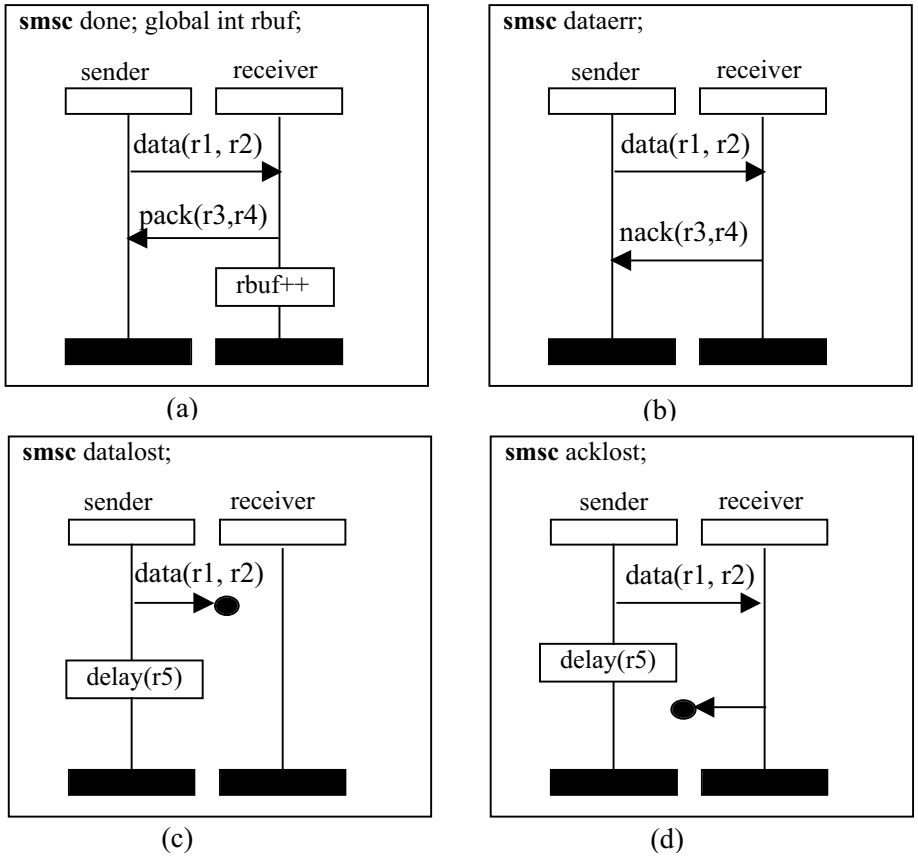


Figure 7. The 4 scenarios of the Stop and Wait protocol.

Figure 8 provides an additional SMSC, **GetFrame**, to specify how the sender acquires data from the sending buffer. This SMSC serves as the protocol starting point. The full behavior of this protocol is described by combining these five SMSCs. Figure 9 shows the composition methods.

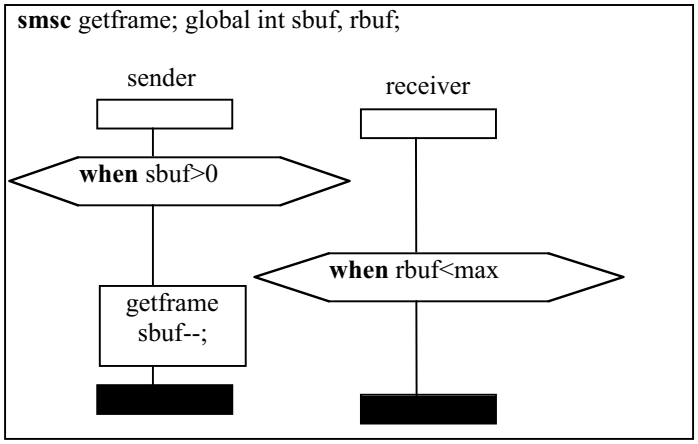


Figure 8. The GetFrame SMSC.

The GetFrame SMSC describes the behavior of the sender when it fetches a data frame from the sending buffer. After a data frame is acquired, the execution proceeds into one of four alternative 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** within this execution, the execution has to loop among the four scenarios indefinitely until the SMSC **done** is selected.

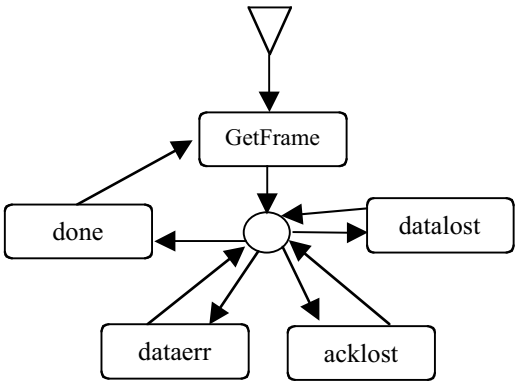


Figure 9. The model of the Stop and Wait protocol.

5.2 Modeling the Data Sending and Receiving Processes

The data sending and receiving processes are modeled as SANs because they are available in the Möbius tool and are suitable for modeling such processes. The SAN model for the sender (i.e., data sending process) is shown in Figure 10. A token in the place *sdata* represents a large block of data, for example a

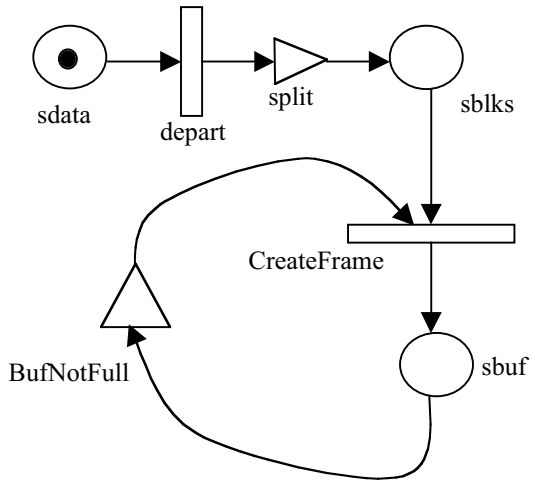


Figure 10. The SAN of the sender.

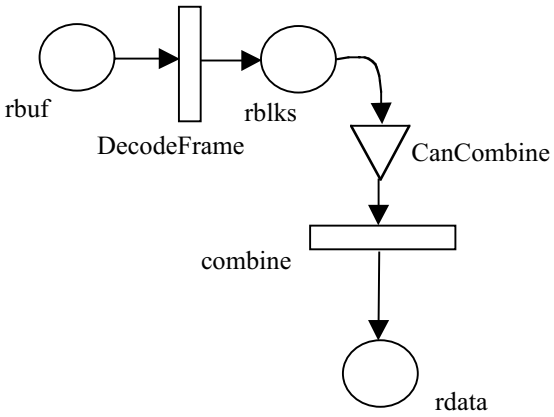


Figure 11. The SAN of the receiver.

file 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. *CreateFrame* can fire if at least one token exists in *sblks* and the predicate of the input gate *BufNotFull* evaluates to true (i.e., indicating the sending buffer is not full). Each time *CreateFrame* fires, a token is deposited into the place *sbuf*. Each token in *sbuf* represents a data frame that will be sent using the stop and wait protocol

(i.e., *subf* represents the sending buffer).

The SAN model for the data receiving process or the receiver is shown in Figure 11. The procedure of processing the received frames is the inverse of what is done by the sending process.

5.3 A Heterogeneous Model of the Whole System

The heterogeneous model can be constructed using the Möbius Join and Replicate mechanism as shown in Figure 12. In Figure 12, the ***sender*** and ***receiver*** refer to the SAN models of the sender and receiver. The word ***protocol*** refers to the SMSC model for the stop and wait protocol.

Before the models can be joined, the shared state variables must be defined. The Möbius *join* construct uses the shared state variable to merge different models together (from either the same or different formalism(s)). In our example, *rbuf* and *sbuf* are shared state variables. In the SAN model, places *rbuf* and *sbuf* are translated into state variables for the Möbius representation. The global data *rbuf* and *sbuf* in the SMSC are also translated into state variables. These state variables are shareable. In fact, they represent the same system components in different models.

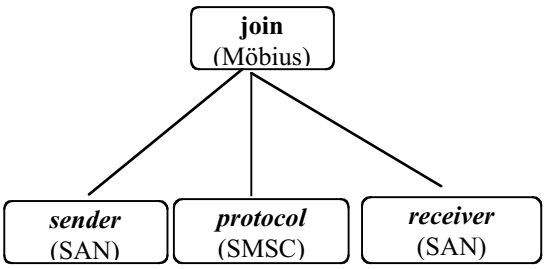


Figure 12. Composing the system model.

5.4 Experimental Result

To show that Möbius can solve an SMSC model, we defined one reward variable to measure the time that the system spends handling error data. Whenever an error occurs in the channel, the sender must 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 the error processing time. We are interested in how the channel error probability and the delay time impact the error processing time. The result of this analysis is shown in Figure 13.

Examining Figure 13, we see that the percentage of time processing errors is roughly proportional to the channel error probability. A higher error probability causes more time spent processing error messages. Error processing time is also affected by delay time. Longer delay times imply that the sender

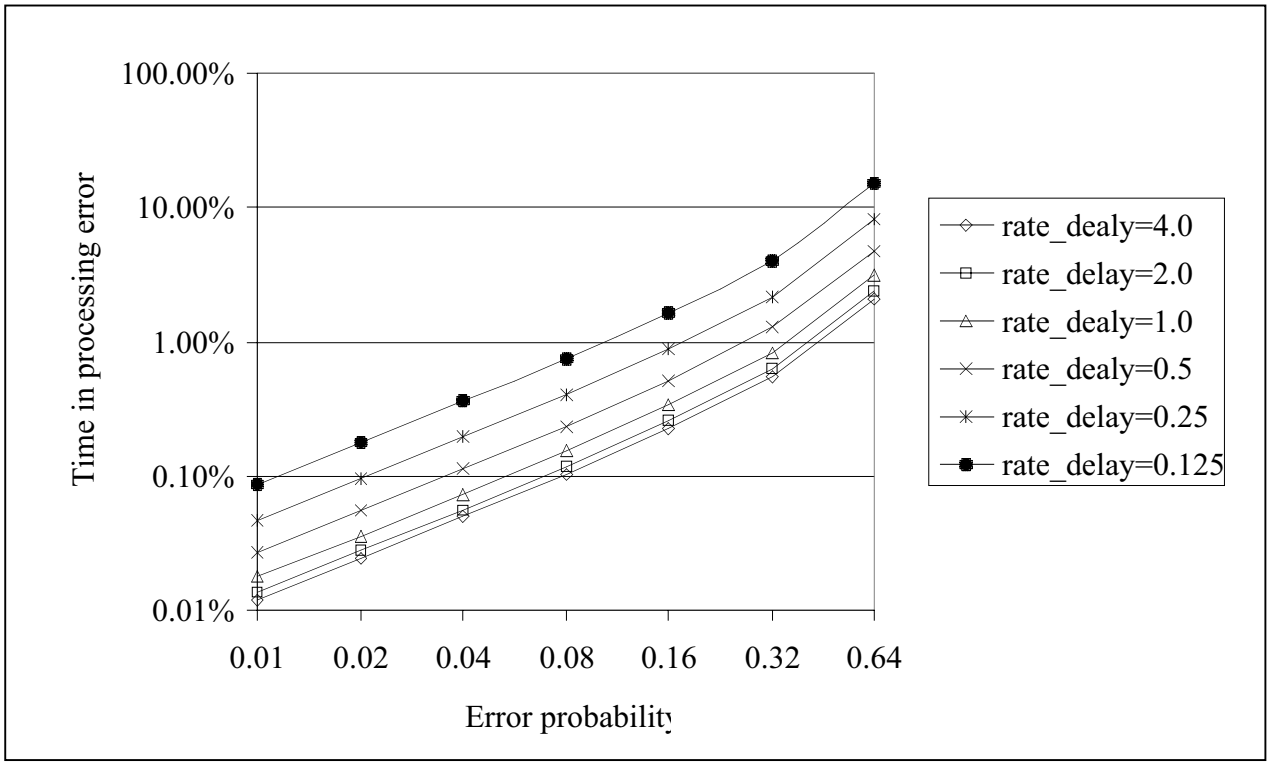


Figure 13. Error processing time of the system.

would have to wait for a longer time to retransmit data. A longer delay time results in a higher percentage of time that the system spends processes errors (note that rate is defined as the inverse of time and higher rate means shorter delay time), i.e., wasted bandwidth.

6. Conclusion and Future Work

The Message Sequence Chart formalism and the Möbius multi-formalism modeling framework were studied. Based on the MSC formalism, we defined a new formalism – Stochastic Message Sequence Chart, an extension to the MSC formalism. SMSC can be used to describe system behavior in the same way as the MSC language. Furthermore, SMSC models contain more information about the system than the corresponding MSC models. By associating each activity with 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 using the newly defined / validated SMSC language.

To integrate SMSC into Möbius, we defined the SMSC state variables and SMSC activities, which correspond to the Möbius state variables and actions, respectively. The C++ classes were implemented used to specify SMSC models. The vertical and alternative 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 realized within the Möbius framework.

The next step in this work would be to implement the UI (user interface, a SMSC graphical editor) within the Möbius framework. The UI should be implemented in Java to make it platform neutral and will enable users to specify SMSC models within the Möbius framework. Eventually, the graphical or textural SMSC models are complied and linked with the Möbius libraries to generate an executable model and the model is either simulated or solved analytically.

Some constructs of the SMSC language, including inline expressions and horizontal compositions, have not been defined within the Möbius framework. Those constructs merely provide shortcuts when specifying the system behavior and do not contribute to the fundamental translations we have defined. However, they should be expressed using the defined SMSC classes and 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 only state variables, an SMSC model may be composed with other models by also sharing activities/actions.

7. References

1. F. Sheldon, G. Xie, O. Pilskalns, and Z. Zhou, *A Review of Some Rigorous Software Design and Analysis Tools.* Software Focus, 2002. 2(4): p. 140-149.
2. B.R. Haverkort and I.G. Niemegeers, *Performability modeling tools and techniques.* Performance Evaluation, 1996. 25(1): p. 17 (24 pages).
3. J. Couvillion, R. Freire, R. Johnson, W.D. Obal, M.A. Qureshi, M. Rai, W. Sanders, and J. Tvedt, *Performability Modeling with UltraSAN.* IEEE Software, 1991. 8(5): p. 69-80.
4. G. Ciardo, R.A. Marie, B. Sericola, and K.S. Trivedi, *Performability Analysis Using Semi-Markov Reward Processes.* IEEE Trans on Computers, 1990. 39(10): p. 1251-1264.
5. J. Rupe and W. Kuo, *Performability of FMS Based on Stochastic Process Models.* Int'l Jr. of Production Research, 2001. 39(Part 1): p. 139-156.
6. J. Magott, *Performance evaluation of communicating sequential processes (CSP) using Petri nets.* IEE Proc. E, Computers and Digital Techniques., 1992. 139(3): p. 237-241.
7. M.K. Molloy, *Performance Analysis Using Stochastic Petri Nets.* IEEE Trans on Computers, 1982. C-31(9): p. 913-917.

8. T. Murata, *Petri Nets: Properties, Analysis and Applications.* Proc. of the IEEE, 1989. 77(4): p. 541-580.
9. J.C.M. Baeten, *Process algebra: special issue editorial.* The Computer Jr., 1994. 37(5): p. 474.
10. ITU-T, *Formal Semantics of Message Sequence Charts*. 1998: Geneva.
11. ITU-T, *Recommendation Z.120: Message Sequence Chart(MSC)*. 1999: Geneva.
12. M.A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis, *Modeling with Generalized Stochastic Petri Nets*. 1994: John Wiley and Sons.
13. 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.
14. J. Hilston and H.U. Hermanns, *Stochastic Process Algebras: Integrating Qualitative and Quantitative Modeling*. 1994, Univ. of Erlangen-Nurnberg: Germany.
15. E.L. Cunter, A. Muscholl, and D.A. Peled, *Compositional Message Sequence Charts.* LNCS, 2001(2031): p. 496-511.
16. W. Damm and D. Harel, *LSCs: Breathing Life into Message Sequence Charts.* Formal Methods in System Design, 2001. 19(1): p. 45-80.
17. B. Finkbeiner and I. Kruger. *Using Message Sequence Charts for Component-based Formal Verification*. In *OOPSLA 2001 Wkshp on Spec. and Verification of CBS*. 2001. Tampa, FL, USA.
18. D. Daly, D.D. Deavours, A.J. Stillman, P.G. Webster, and W.H. Sanders, *Mobius: An Extensible Tool for Performance and Dependability Modeling.* LNCS, 2000(1786): p. 332-336.
19. W. Sanders, Obal, W., Qureshi, A., and Widjanarko, F., *The UltraSAN Modeling Environment.* Performance Evaluation, 1995. 24(1): p. 89-115.
20. J. Hillston, *A Compositional Approach to Performance Modeling*. 1996: Cambridge Univ. Press.
21. G. Clark and W.H. Sanders. *Implementing a Stochastic Process Algebra within the Möbius Modeling Framework*. In *Process Algebra and Probabilistic Methods: Performance Modeling and Verification: Proceedings of the Joint International Workshop, PAPM-PROBMIV 2001*. 2001. RWTH Aachen, Germany: Berlin: Springer.
22. 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.
23. G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J.M. Doyle, W.H. Sanders, and P. Webster. *The Möbius Modeling Tool*. In *Proceedings of the 9th International Workshop on Petri Nets and Performance Models*. 2001. Aachen, Germany.
24. A.S. Tanenbaum, *Computer Networks*. 3rd ed. 1996: Prentice-Hall.