

Möbius : An Extensible Framework For Performance and Dependability Modeling *

David Daly, Daniel D. Deavours, Jay M. Doyle,
Aaron J. Stillman, Patrick G. Webster, and William H. Sanders
Department of Electrical and Computer Engineering
Coordinated Science Laboratory

University of Illinois at Urbana-Champaign
1308 W. Main St., Urbana, IL, U.S.A.

{ddaly, deavours, jmdoyle, astillma, patweb, whs}@crhc.uiuc.edu
<http://www.crhc.uiuc.edu/PERFORM>

Abstract

Möbius is a framework for building extensible modeling tools that support model specification in multiple modeling formalisms as well as many different model solution methods. Heterogeneous models appear to be the only reasonable approach to modeling large systems that cross over many different application domains. The key to the Möbius tool is an abstract functional interface through which the models can communicate with other models and solvers. We have implemented a tool using the Möbius framework and have shown that sophisticated atomic, composed, and reward model formalisms can be realized in the framework.

1 Introduction

1.1 Motivation

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

Just as we concluded that there is no single modeling formalism appropriate for all applications, we can also say that there is no single solution method that is best for all models. Simulation-based solution techniques allow for more flexibility in model specification, but have a harder time capturing the effects of rare but important events. Also, solution via simulation often requires a long amount of time in order to ensure confidence in the model's solutions. Analytical-based techniques provide exact solutions for reward metrics, but have many model specification restrictions. For instance, many analytical solution methods are based on solving Markov or semi-Markov processes. A Markov process representation requires that state changes be exponentially distributed. In many cases this may not reflect the true operation of the modeled system, and even if it does, the size of the model's state space may make analytical solution infeasible.

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

*This material is based upon work supported by DARPA/ITO under Contract No. DABT63-96-C-0069. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA/ITO.

for each part of a model and later combine the models together to form one large, heterogeneous model.

1.2 Möbius

We have designed our framework and tool around this concept of heterogeneous modeling. The tool implements an extensible framework by using an abstract functional interface. This abstract functional interface provides both intermodel communication and communication between models and solvers. This precisely defined abstract functional interface defines the requirements for developing new formalisms and solvers that work within the framework.

The functional interface allows the formalism to decide how it wants to store and change its state. Since other parts of the model (and solvers) need not know anything about the inner workings of a formalism, the formalism implementor is free to make many optimizations. Another possible method for implementing a heterogeneous modeling environment would be to translate all models down to a universal representation. However, this translation process would undoubtedly remove formalism-specific knowledge, resulting in less efficient model solutions.

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

2. Möbius Framework

2.1. Overview

We begin with a brief overview of the concept of a model in the Möbius framework. The Möbius framework is a very general way to specify a model. We define a *formalism* as a language for expressing a model within the Möbius framework, frequently using only a subset of the options available within the framework.

The Möbius framework is a powerful and flexible way to specify models. A *model* is a collection of state variables, actions, groups, and reward variables. Briefly, state variables hold the state information of the model, actions

change the state of the model over time, a group is a collection of actions that coordinate behavior in some specific way, and reward variables are a way of measuring something of interest about the model.

Although the basic elements of a model are very general and powerful, formalisms need not make use of all the generality. In fact, it may be useful to restrict the generality in order to exploit some property for efficiency. The purpose of some formalisms is to expose these properties easily, and to take advantage of them for efficient solution. Möbius was designed with this in mind.

For convenience, it is useful to classify models into certain types. The most basic category is that of atomic models. An atomic model is a self-contained (but not necessarily complete) model that is expressed in a single formalism. Several models may be structurally joined together to form a single larger model, which is called a *composed model*. Naturally, a composed model is a model, and may itself be composed. A model that is more loosely connected by sharing solutions is called a *connected model*. Next, we describe each of these concepts in greater detail.

2.2. Model Definition

A model is made up of state variables, actions, groups, and reward variables. We begin by describing state variables.

State Variables A *state variable* contains some state information about the model. Generally, the *state* of a state variable is its type and its value. The type of a state variable specifies the range of values in which the value of a state variable may take. For example, state variable s may have a type of "integer." The value of s may therefore be, for example, 5.

Basic types of state variables may be integers, reals, or names of other state variables (useful for a pointer-like facility). Types may be subsets of basic types, and they may also be ordered or unordered sets of types. By this recursive definition, types may be quite simple, as in Petri net places, or complex data structures.

Actions Actions are the mechanism by which the state of state variables changes over time. For brevity, we omit a formal definition of actions here, and describe briefly how actions work. Actions work much like transitions of GSPNs [4] or activities of SANs [3], except that they are in many ways more general. We describe ways in which they are different.

Actions may work according to one of three work policies: race-enable, race-age, or race-resample [5]. Actions and transitions typically have the race-enable work policy.

Race-enabled actions choose a completion time each time they become enabled. Actions that are race-resample will pick new completion times at every state change in which they are enabled. Actions that are race-age will in some way remember how close to completion they are, so that if they become disabled before they fire and at some later time become enabled again, they will be proportionally closer to completion. A state-dependent mixture of policies is also allowed.

Actions may have any state-dependent state-changing function, any delay function, and any of the described execution policies. Formalisms may place useful restrictions on these generalities.

Groups Groups are a set of actions that may coordinate in some ways. Preselection groups, [5] both variable and persistent, are supported. In addition, a new type of group called postselection is supported to mimic action-case behavior in SANs.

Reward Variables Work by Obal and Sanders [2] has defined a type of reward variable that embeds a state machine into the reward structure. Möbius uses this concept in defining a more general, state machine-based reward variable concept. This way of specifying reward variables is very general and powerful, and encompasses previously known reward-variable specification formalisms. The rate-impulse reward structure of SANs [6] is an example of the reward formalisms that may be expressed by this more general reward variable structure.

Results Results are not a part of the definition of a model, but it is convenient to describe them here. A result is some solution to a reward variable. For an analytic solution to a reward variable, for example, the result may include the expected value, the variance, perhaps a distribution, the type of solver used, the stopping criterion, and if possible, the error bounds. For the same reward variable, a simulation result may include the expected value, the variance, the confidence interval and width of the expectation and variance, and the number of observations.

Results form a database from which information may be retrieved and plotted, or used to provide information for a parameter to another model, as is the case for connected models (see Section 2.3).

2.3. Model Classification

Complex models are usually built out of modules of smaller models. The first step in building a model is to build an atomic model. An atomic model is the simplest type of model which is built using a single formalism. If

the atomic model has reward variables, it is called a reward-atomic model; otherwise it is called a simple-atomic model. Examples of atomic model formalisms include stochastic activity networks [3], generalized stochastic Petri nets [4], and queuing networks.

Typically (but not necessarily), atomic model formalisms specify simple-atomic models. A reward formalism is a formalism that takes simple-atomic models and produces reward-atomic models. This allows developers to develop a language for specifying reward variables that may apply to any atomic model formalisms.

A composed model is a collection of models that have been combined structurally to make up a single new model. Typically, models are composed by sharing state variables or actions. Sometimes, by combining models according to some simple rules, efficiencies may be exposed in in state-space reduction or model solution. Examples of composed model formalisms are the Rep-Join formalism of SANs [7], graphical composition of [1], and synchronizing transitions.

Models that are more loosely connected are called connected models. A *connected model* is a collection of models in which there is an explicit dependency between models; the result of one model is used as a parameter to another model. If the dependency graph is cyclic, some form of fixed-point iterative technique may be used, depending on the particular connected model formalism. We anticipate that Möbius will be a great aid in performing research and implementing connection formalisms in the future.

3 Möbius Tool

The first step in implementing the Möbius tool was to define the abstract functional interface that is at the core of the tool. This functional interface is a software realization of the Möbius framework. We realized the functional interface as a set of C++ base classes from which all models must be derived. We defined the functional interfaces as pure virtual methods. This forces any formalism implementor to define the operation of all the methods in the functional interface. In the same fashion, we constructed C++ base classes for other Möbius framework components, including actions, groups, and state variables (see Section 2.2). Each of these entities also has methods that are part of the abstract functional interface.

The Möbius tool architecture (see Figure 1) is separated into two different logical layers: model specification and model execution. All model specification in our tool is done through Java graphical user interfaces, and all model execution is done exclusively in C++. We decided to implement the executable models in C++ for performance reasons. For every formalism there is a separate editor. Editors produce compilable C++ code as output so that the final ex-

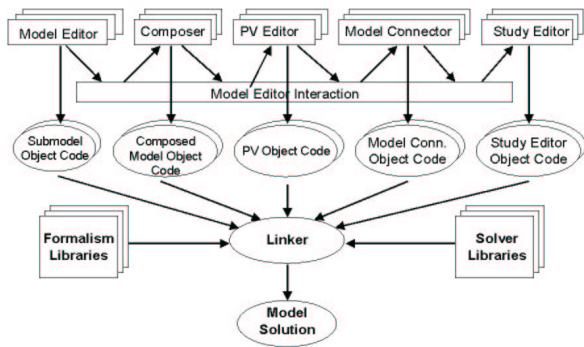


Figure 1. Möbius Architecture

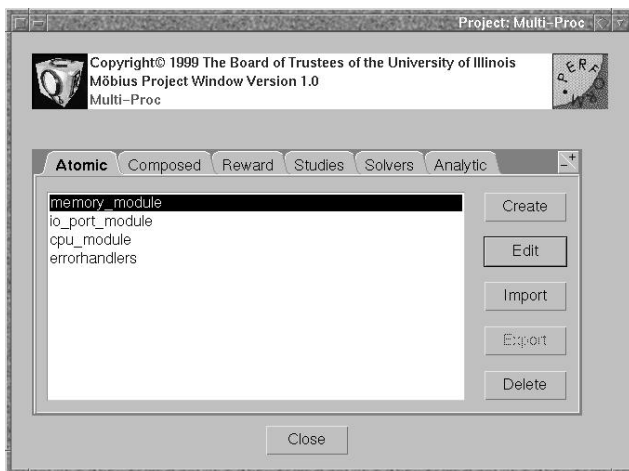


Figure 2. Möbius Project Window

ecutable model is entirely specified within C++. The C++ files produced by the editor are compiled and the tool links the object code with formalism libraries and solver-specific libraries.

After the abstract functional interface had been specified, we implemented an atomic, composed, and reward model formalism within the abstract functional interface. This first set of formalisms proves that sophisticated modeling formalisms can be integrated into an extensible modeling tool. Because of our past work with *UltraSAN*, we chose to reimplement *UltraSAN*'s atomic, composed, and reward model formalisms inside the Möbius tool.

Currently, our tool contains the following model specification components:

Project Manager This is a utility that allows a user to group related models into a single, logical project

(see Figure 2). A project may contain many atomic, composed, reward, and connected models.

SAN Editor This is an atomic model editor in which the user can specify models using the stochastic activity network formalism (see Figure 3) [3].

Replication-Join Composer Model Editor This editor allows the user to specify a composed model by using two composed model constructs: join and replication. The join construct combines multiple models via a notion of shared state. The replication construct allow the modeler to replicate a portion of the model n times. The n instances of the replicated model are connected together through a set of common state variables [7].

Rate-Impulse Reward Editor This editor allows the user to specify reward variables whose value is determined by a set of state-based rate and impulse functions [6].

Study Editors Through all phases of model specification, global variables can be used as input parameters. Study editors allow the modeler to specify the value of those global variables. The value of global variables may be specified over a range. The output of these study editors is a set of experiments (each experiment is a unique specification of global variable values).

Currently, our tool contains the following model solution components:

Discrete Event Simulator This generic simulator allows any model to be simulated for transient or steady-state reward measures. It also allows the simulation to be distributed across a heterogeneous set of workstations, resulting in a near-linear speed-up.

State-Space Generator This module creates a Markov process description for models that have exponentially distributed state changes. The result of the state-space generator is used as an input for many different analytical solvers.

Analytical Solvers There are several analytical solvers implemented in the Möbius tool. They include transient and steady-state solvers.

In the process of developing these first Java interfaces, we constructed several Java class packages that facilitate the construction of graphical user interfaces for the Möbius tool. Having such utilities should minimize the amount of time required to implement a specification module for a new formalism or solution technique.

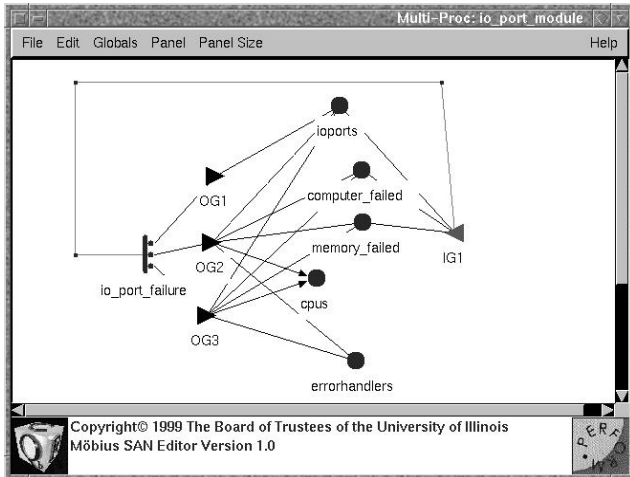


Figure 3. Möbius SAN Editor

4 Future Directions

The next important step in the development of the Möbius tool will be to implement more formalisms in it to prove that it is truly an extensible architecture. There are many different atomic model formalisms that could be implemented, including queuing networks, GSPNs, reliability block diagrams, and fault trees. We will implement a new composed model formalism in the tool to allow generic, graph-interconnect composition. Such composition can be used to exploit existing symmetries within a model's state space [1]. There also exists a great deal of opportunity to explore connection formalisms.

We also plan to store all solver results into a results database. The results database will be coupled with a results browser capable of submitting sophisticated queries. The results browser will allow a modeler to create detailed reports of model results. With the results database, one will be able to look at the results from different model versions across multiple solution techniques.

Since the default format for reports and model documentation in Möbius is HTML, we plan to add functionality in Möbius that can launch an application to view HTML output.

5 Acknowledgments

We would like to acknowledge the work done by former members of the Möbius group: G. P. Kavanaugh, J. M. Sowder, and A. L. Williamson.

References

- [1] W. D. Obal II. *Measure-Adaptive State-Space Construction Methods*. PhD thesis, University of Arizona, 1998.
- [2] W. D. Obal II and W. H. Sanders. State-space support for path-based reward variables. In *Proceedings of the 1998 International Computer Performance and Dependability Symposium (IPDS '98)*, pages 228–237, Durham, North Carolina, September 1998.
- [3] J. F. Meyer, A. Movaghar, and W. H. Sanders. Stochastic Activity Networks: Structure, Behavior, and Application. In *Proceedings of the International Conference on Timed Petri Nets*, pages 106–115, Torino, Italy, July 1985.
- [4] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modeling with Generalized Stochastic Petri Nets*. Series in Parallel Computing. John Wiley & Sons, 1995.
- [5] Marco Ajmone Marsan, Gianfranco Balbo, Andrea Bobbio, Giovanni Chiola, Gianni Conte, and Aldo Cumani. The Effect of Execution Policies on the Semantics and Analysis of Stochastic Petri Nets. *IEEE Transactions on Software Engineering*, 15(7), July 1989.
- [6] W. H. Sanders and J. F. Meyer. A unified approach for specifying measures of performance, dependability, and performability. In A. Avizienis, H. Kopetz, and J. Laprie, editor, *Dependable Computing for Critical Applications, Vol. 4 of Dependable Computing and Fault-Tolerant Systems*, pages 215–237. Springer-Verlag, 1991.
- [7] W. H. Sanders and J. F. Meyers. Reduced base model construction methods for stochastic activity networks. *IEEE Journal on Selected Areas in Communications, special issue on Computer-Aided Modeling, Analysis, and Design of Communication Networks*, 9(1):25–36, Jan. 1991.