Möbius : An Extensible Tool For Performance and Dependability Modeling *

David Daly, Daniel D. Deavours, Jay M. Doyle, Aaron J. Stillman, Patrick G. Webster
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}@crhc.uiuc.edu

1 Introduction

Möbius is a tool for building performance and dependability models of stochastic, discrete-event systems. Möbius makes validation of large dependability models possible by supporting many different model solution methods as well as model specification in multiple modeling formalisms.

The motivation for building the Möbius tool is based upon the observation that no formalism has shown itself to be the best for building and solving models across many different application domains. Similarly, no single solution method is appropriate for solving all models. Furthermore, new techniques in model specification and solution are often hindered by the necessity of building a complete tool every time a novel concept is realized. We deal with these three issues by defining a broad framework in which new modeling formalisms and model solution methods can be easily integrated. A *modeling framework* is a formal, mathematical, specification of model construction and execution. Our modeling tool defines an abstract functional interface that facilitates intermodel communication as well as communication between models and solvers. This abstract functional interface also allows the modeler to specify different parts of the model in different formalisms.

2. Möbius Framework

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. State variables may be simple integers, as in Petri net places, or complex structures. Actions change the state of the model over time. They may have a general delay distribution or a general state-change function, and may operate by any one of several execution policies. A *group* is a collection of actions that coordinate behavior in some specific way. *Reward variables* are ways of measuring something of interest about the model. They embed a state machine to allow path-based reward variables.

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 how we implement this framework as a tool.

3 Möbius Tool

The first step in implementing the Möbius framework was to define the abstract functional interface that is at the core of the tool. 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. 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. Every formalism has a separate editor for specifying a particular piece of the model. Editors produce compilable C++ code as output so that the final executable 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.

*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.

©1999, David Daly, Daniel D. Deavours, Jay M. Doyle, Aaron J. Stillman, Patrick G. Webster

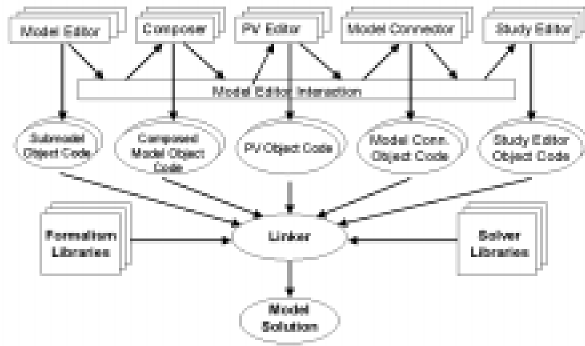

Figure 1. Möbius Architecture.

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

Currently our tool contains the following model specification

SAN Editor An atomic model editor in which the user can specify models using the stochastic activity network formalism [2].

Replication-Join Composer Model Editor This editor allows the user to specify a composed model by using two composed model constructs: join and replication [4].

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 [3].

Study Editors Through all phases of model specification, global variables can be used as input parameters. These editors allow the modeler to specify the value of those global variables.

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 both 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.

4 Future Directions

The next important step in the development of the tool will be to implement more formalisms in the Möbius framework 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 opportunities 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. This will allow a modeler to create detailed reports of model results. With the results database, one can look at the results from different model versions across multiple solution techniques. The default format for model documentation and report generation will be HTML. The user will have the ability to launch an application to view the HTML output from the tool.

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] 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.

[3] 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.

[4] 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.