Möbius: An Extensible Tool For Performance and Dependability Modeling [⋆]

David Daly, Daniel D. Deavours, Jay M. Doyle,
Patrick G. Webster, and William H. Sanders

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory and
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
1308 W. Main St., Urbana, IL, U.S.A.
{ddaly, deavours, jmdoyle, patweb, whs}@crhc.uiuc.edu
www.crhc.uiuc.edu/PERFORM

1 Introduction

Möbius is a system-level performance and dependability modeling tool. 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 was 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. In this context, a *modeling framework* is a formal, mathematical specification of model construction and execution. In implementing the framework we define an *abstract functional interface* [1], which is realized as a set of functions 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 concepts of a formalism and a model in the Möbius framework. The Möbius framework provides a very general way to specify a model in a particular formalism. 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.

We define models within the Möbius framework using a few basic concepts. A *model* is a collection of state variables, actions, groups, and reward variables

[⋆] This material is based upon work supported by DARPA/ITO under Contract No. DABT63-96-C-0069 and the National Science Foundation's Next Generation Software Initiative.

expressed in some formalism. Briefly, *state variables* hold the state information of the model. State variables may be simple integers, as in Petri net places, or complex data structures. *Actions* change the state of the model over time. They may have a general delay distribution and 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 a component of a larger composed model. A model that is more loosely connected by the sharing of 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 is to define the abstract functional interface that is at the core of the tool. We have implemented the functional interface as a set of C++ base classes from which all models must be derived. In doing so, we define the functional interfaces as pure virtual methods. This requires that any formalism implementor define the operation of all the methods in the functional interface. In the same fashion, we construct C++ base classes for other Möbius framework components, including actions, groups, state variables, and reward 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 specified entirely 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 was specified, we implemented an atomic model formalism, two composed model formalisms, and a reward model formalism inside the Möbius framework. 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; we also implemented a new composed model editor.

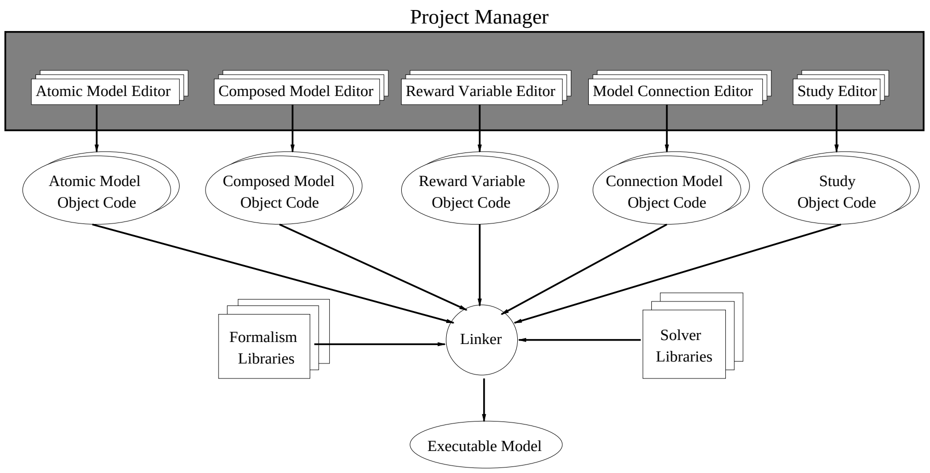

Fig. 1. Möbius Architecture.

Currently, our tool contains the following model specification editors:

SAN Editor An atomic model editor in which the user can specify models using the stochastic activity network formalism [3].

Replication-Join Composed Model Editor This editor allows the user to specify a composed model by using two composed model constructs: replicate and join [5].

Graph Composer Editor This editor allows the user to construct a composed model through an arbitrary graph of submodels connected through shared state [2].

Rate-Impulse Reward Editor This editor allows the user to specify reward variables whose values are determined by a set of state-based rate and impulse functions [4].

Study Editors Through all phases of model specification, global variables can be used as input parameters. These editors allow the modeler to specify the values 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 a model that has exponentially distributed delays between state changes. The output 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 show 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, stochastic process algebras, and fault trees. There are also many opportunities to explore connection formalisms and model solution methods that use specific knowledge of reward measures to reduce the cost of solution.

We also plan to store all solver results in 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, a user will be able to look at the results from different model versions across multiple solution techniques. A visualization tool will also be provided to display model results visually. The default format for model documentation and report generation will be HTML. The user will have the ability to launch an application form the tool to view the 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, A. J. Stillman, and A. L. Williamson.

References

1. Jay M. Doyle. Abstract model specification using the möbius modeling tool. Master's thesis, University of Illinois, January 2000.
2. W. D. Obal II. *Measure-Adaptive State-Space Construction Methods.* PhD thesis, University of Arizona, 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. 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.
5. 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.