The Möbius Modeling Tool [*]

Graham Clark, Tod Courtney, David Daly, Dan Deavours,
Salem Derisavi, Jay M. Doyle, William H. Sanders, and Patrick Webster

Dept. of Electrical and Computer Engineering and Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 W. Main St., Urbana, IL, U.S.A.

{gcla,tod,ddaly,deavours,derisavi,jmdoyle,whs,patweb}@crhc.uiuc.edu
http://www.crhc.uiuc.edu/PERFORM

Abstract

Despite the development of many modeling formalisms and model solution methods, most tool implementations support only a single formalism. Furthermore, models expressed in the chosen formalism cannot be combined with models expressed in other formalisms. This monolithic approach both limits the usefulness of such tools to practitioners, and hampers modeling research, since it is difficult to compare new and existing formalisms and solvers. This paper describes the method that a new modeling tool, called Möbius, uses to eliminate these limitations. Möbius provides an infrastructure to support multiple interacting formalisms and solvers, and is extensible in that new formalisms and solvers can be added to the tool without changing those already implemented. Möbius provides this capability through the use of an abstract functional interface*, which provides a formalism-independent interface to models. This allows models expressed in multiple formalisms to interact with each other, and with multiple solvers.*

1 Introduction

Many performance/dependability modeling formalisms and model solution methods have been developed. Determination of which particular formalism and model solution method are "best" for a particular project depends on the type of system being modeled, the level of detail required to support the chosen measures, the time available for the study, computational resources, and more. In short, it is clear that no single modeling formalism or solution method is best in all circumstances. Furthermore, in a large modeling project it may be desirable to express different parts of a model in different formalisms, and permit the parts to interact or synchronize. In fact, it is often the case in a large modeling project that different parts of a model are built by different people with different backgrounds at different times. Techniques and tools are needed to combine these models and/or results into a single, coherent model.

In spite of this issue, most modeling tool implementations have been monolithic (with a few notable exceptions, see below), in the sense that they support a single modeling formalism and one or more solution methods, and models expressed in the chosen formalism cannot be combined with models expressed in other formalisms [1]. This state of affairs both limits these tools' usefulness to practitioners, and hampers modeling research, since it is difficult to compare new and existing formalisms and solvers. Modeling tools are clearly needed that support multiple interacting formalisms and solvers, and are extensible so that new formalisms and solvers can be added without requiring changes to existing components.

We are not the first to recognize this need. An early and important effort to combine multiple modeling formalisms into a single tool is SHARPE [2]. In SHARPE, models can be expressed in a variety of modeling formalisms, and results can be exchanged between submodels in the form of exponential-polynomial distribution functions. SMART [3], another software tool that integrates multiple modeling formalisms, was developed more recently. SMART supports the analysis of models expressed as stochastic Petri nets and queuing networks, and is implemented in a way that permits the easy integration of new solution algorithms in the tool. Models interact in SMART by exchanging results, possibly repeatedly, using fixed-point iteration. The DEDS (Discrete Event Dynamic System) toolbox also integrates multiple modeling formalisms, but does so by converting models expressed in different formalisms into a common "abstract Petri net notation" [4]. Each of these tools was an important step forward in building multi-formalism tools. The next step in generality would be to build a modeling framework without presup-

[*] This material is based upon work supported in part by the National Science Foundation under Grant No. 9975019 and by the Motorola Center for High-Availability System Validation at the University of Illinois (under the umbrella of the Motorola Communications Center).

posing what types of modeling formalisms would be supported or what methods would be used to combine submodels. This goal guided us in the development of the Möbius framework and tool.

Möbius was inspired by past attempts to integrate multiple modeling formalisms and solvers, and provides an infrastructure to support multiple interacting modeling formalisms and solvers. It is also extensible in that new formalisms and solvers can be added to the tool without changing existing formalisms or solvers. It provides this capability through the use of an *abstract functional interface* (AFI [5]) that uses abstract classes to provide a formalism-independent interface that allows models expressed in multiple formalisms to interact with one another and with multiple solvers.

In this paper, we describe the AFI and how the Möbius tool makes use of it to provide a multi-formalism multi-solution modeling environment. Section 2 introduces the AFI, and describes the architecture of the Möbius tool. Section 3 illustrates how we implemented different styles of model formalism in Möbius by making use of the AFI. Section 4 discusses how a modeler proceeds from a model formalism to a set of experiments that generate performance measures, and Section 5 discusses how Möbius is used to solve models. Section 6 concludes the paper.

2 Enabling Multi-Formalism Multi-Solution Modeling

The Möbius project was inspired by previous work on *UltraSAN* [6], which was developed to model and solve stochastic activity networks (SANs [7]). Möbius is broader in scope than *UltraSAN* and we aim to provide support for many different modeling formalisms. These include traditional performance modeling paradigms such as networks of queues and stochastic Petri nets (SPNs), newer approaches such as stochastic process algebras (SPAs) and stochastic automata networks [8], and more specialized approaches such as fault trees and combinatorial block diagrams.

Our design goal for the Möbius tool was extensibility, meaning that it should be possible to add new modeling formalisms and, to a large degree, that they should be able to interact with existing formalisms and solvers without requiring that the tool undergo any changes. Extensibility also means that it should be easy to add new model solvers so that, to the extent theoretically possible, they can be employed to solve models built with existing and future formalisms. The *Möbius framework* [1, 9] is the theoretical underpinning of the Möbius tool, and the key to this extensibility. The framework specifies a set of modeling components, and their interactions, distilled from the features present in many different modeling paradigms. Under the framework, model formalism components are mapped to Möbius components.

In the Möbius tool, the framework is present as the abstract functional interface, which uses abstract classes to implement Möbius components. While the current AFI does not implement all the framework components described in [9] (for historical reasons), it is similar in scope and capability. If a model formalism is to be implemented, its various components must be presented as classes derived from the Möbius abstract classes. Other model formalisms and model solvers are then able to interact with the new formalism by accessing its components through the Möbius abstract classes. The set of distilled model components captured in these abstract classes constitutes the AFI.

2.1 Architecture

The various components of the Möbius tool are divided into two categories: model specification components and model solution components. Models are specified through a front-end, which consists of a set of graphical user interfaces running within a main Java interface. The tool is organized as a series of editors, classified according to model types. Each formalism or solver supported by Möbius has a corresponding editor in the main interface. The Möbius tool is designed so that new formalisms can be implemented and employed if they adhere to the AFI. To accommodate new formalisms, the Möbius application loads every formalism-specific editor dynamically from a Java archive (`jar` file). This design allows new model editors to be incorporated without modification of the existing code, supporting the extensibility of the Möbius tool.

Models can be solved either analytically/numerically or by simulation. From each model, C++ source code is generated and compiled, and the object files are linked together to form a library archive. These libraries are linked together along with the Möbius base libraries to form the executable for the solver. The Möbius base libraries implement the base components of the particular model formalism, the AFI, and the solver algorithms. The organization of Möbius components to support this model construction procedure is shown in Figure 1.

The front-end is used to launch the solvers and view the generated results. For example, if a simulation is being conducted, the model is compiled and linked to the Möbius simulation libraries, and the front end may launch multiple processes on multiple remote and local machines. Each executable simulates the model and reports its observations back to the front-end using UNIX sockets. The AFI views models as consisting of two sets of components: state variables, which store model state, and actions, which change model

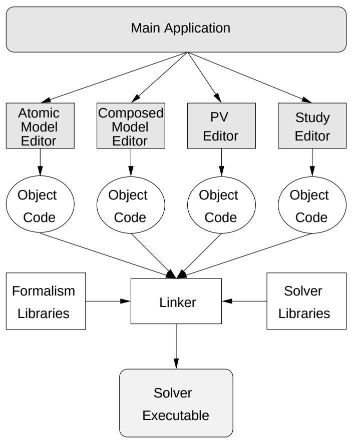

Fig. 1
MÖBIUS TOOL ARCHITECTURE

int StateSize(): Returns the number of bytes used to represent the state variable.
void SetState(void*): Changes the state of the state variable.
void CurrentState(void*): Copies the state of the state variable to the specified memory location.
List⟨BaseAction⟩* AffectingActs(): Returns a list of actions whose completion may alter the value of this state variable.
List⟨BaseAction⟩* EnablingActs(): Returns a list of actions whose enabling conditions depend on the value of this state variable.

TABLE I
SELECTION OF MÖBIUS AFI FUNCTIONS FOR STATE VARIABLES

state. The Möbius framework [9] specifies additional characteristics of models, such as *properties*, that are used to preserve specialized model features that may be exploited in composition or solution (such as the property that allows a model to feature only exponentially distributed activities). The core of the AFI is built from three base classes: one class for state variables, one for actions, and one defining behavior that the model as a whole should be able to provide. Each of these classes defines a minimal set of methods that are used by Möbius when building composed models, specifying reward variables, and solving models. Next, some key AFI methods are presented and explained.

2.2 State Variables

The base class for Möbius state variables, BaseSV, specifies the minimal interface that any Möbius state variable must provide. A selection of the AFI functions associated with state variables is listed in Table I (a complete list of AFI functions can be found in [5]). A state variable is used to hold a portion of the state of a model as a whole, and thus provides methods, **SetState** and **CurrentState**, for changing and copying that state respectively. The methods **AffectingActs** and **EnablingActs** link state variables with Möbius actions. **AffectingActs** is a method that returns those model actions that by their completion may change the value of this particular state variable. In contrast, **EnablingActs** returns a list of activities whose enabling conditions may depend on the value of this state variable. These methods are useful because in practice, model events often have only a "local" impact on model state. The model formalism implementor may be able to implement these methods intelligently such that depending on the model, only a subset of model actions will be linked to any state variable. These methods are then employed by the Möbius solvers to produce results efficiently.

bool Enabled(): A Boolean expression that determines whether the action is enabled in the current model state.
BaseAction* Fire(): A function that defines the state change if the action completes in the current state.
double SampleDistribution(): A function that returns a sample from the action's time-to-completion distribution in the current state.
int Rank(): A function that provides an integer that is used for priority-based scheduling policies.
double Weight(): A function that returns a real number that is used for probabilistic scheduling policies.
bool ReactivationPredicate(): A function that defines whether the action can restart in the current state.
bool ReactivationFunction(): A function that defines whether a restartable action does restart in the current state.
List⟨BaseSV⟩* AffectedSVs(): A function that returns the set of state variables whose values may be affected by the action's completion.
List⟨BaseSV⟩* EnablingSVs(): A function that returns the set of state variables whose values affect whether or not the action is enabled.

TABLE II
SELECTION OF MÖBIUS AFI FUNCTIONS FOR ACTIONS

2.3 Actions

The base class for Möbius actions, BaseAction, determines the characteristics of model actions. Actions are solely responsible for changing model state. Table II lists a selection of the AFI functions associated with Möbius actions. Möbius models are assumed to evolve in continuous time, and for that reason, each Möbius action takes a stochastically timed period to complete. Since an action is responsible for changing the state of the model, a method **Fire** must be implemented for every action in a given model. The method **Enabled**, when evaluated in the current state, returns a Boolean value that determines whether the action is enabled, meaning it will complete at some point in the future if uninterrupted. In order to determine the time after which an enabled action may complete, the **SampleDistribution** method is used. It does not presuppose the form of the distribution from which the sample is taken; Möbius supports any general probability distribution, such as Weibull and Erlang-k, as well as frequently used distributions like the exponential. Note that Möbius features zero-time *instantaneous* actions, for which **SampleDistribution** returns the value 0.0.

ReactivationPredicate and **ReactivationFunction** are more subtle and allow the expression of different *execution policies*. Such policies have been studied in the literature, in particular for Petri net-based models [10], and have been generalized in the Möbius framework (see [11], for example). The Möbius tool currently implements a subset of this theory. To illustrate the definition of an execution policy, consider a model in which two actions are enabled at a point in time. Since the **SampleDistribution** method (and thus action distributions) may be state-dependent, the completion of one action may lead to a state change that should be reflected in the distribution of the other action. Two available options are either to let the unfinished action continue, or to restart it with a new sample from its (perhaps now modified) distribution. **ReactivationPredicate** should return *true* if the action was allowed to be restarted in the state in which it was originally enabled. **ReactivationFunction** is then used by the Möbius solvers in the newly updated state to determine whether the action should restart. Actions are linked back to state variables with the two methods **AffectedSVs** and **EnablingSVs**.

Rank and **Weight** are methods that assign values to actions that may be used to determine a "winner" from a set of competing actions. These methods are employed in the implementation of BaseGroup, a class that inherits from BaseAction. Intuitively, BaseGroup represents a group of actions from which one is selected as a representative. Möbius supports both pre-selection and post-selection of actions. Pre-selection is used to select a representative action from the group on a state change; this action may then be enabled along with a set of others present in the model. Post-selection is used to select a representative action from the group if the group itself, considered as an action (which is legitimate by inheritance), completes in the current state. More details are available in [5].

2.4 The Model

Möbius specifies that each model must itself implement an interface. Table III lists some of the methods that must be provided for any model of a formalism implemented in Möbius. BaseModel, the base class for Möbius models, provides functions that are used in both model composition and model solution.

void SetState(void*): Sets the state of the model to that read from a specified memory location.
void CurrentState(void*): Writes the current state of the model to a specified memory location.
bool CompareState(void*, void*): Determines whether two state variables hold the same value.
List⟨BaseAction⟩* ListActions(): Returns a list of all model actions.
List⟨BaseSV⟩* ListSVs(): Returns a list of all model state variables.

TABLE III
Selection of Möbius AFI functions for models

First, BaseModel specifies functions that must return the list of all state variables (**ListSVs**) and actions (**ListActions**) contained within the model. Model composition formalisms can use these functions to examine the underlying structure of models. The model base class also has functions similar to those listed earlier for state variables; for example, **CurrentState** is responsible for copying the state of the entire model to a specified memory location.

To show examples of the use of the AFI, the next section focuses on the realization of two types of atomic model formalisms.

3 Building Model Formalisms

Möbius uses an explicit component-based system for describing models. This gives a good match to component-based modeling formalisms, including SPNs, SANs, Markov processes, and queuing networks. We focus on showing how arbitrary Petri net-based models are mapped into the AFI. The AFI has also proven to be general enough to implement a stochastic process algebra formalism, namely PEPA, and we also summarize how this is done.

3.1 Realizing Petri Net-Style Languages

We begin to illustrate the mapping from Petri net formalisms to the AFI with one of the simplest formalisms, stochastic Petri nets.

3.1.1 Stochastic Petri nets

We consider an SPN classically, as a bipartite directed graph consisting of *places*, *transitions* with associated exponentially distributed random variables, and connecting arcs. Figure 2 presents a simple SPN model of a queue with a population size of K. An arriving customer is destined for one of two bins, and the service center requires a customer from each bin. In order to map an SPN to the Möbius AFI, the most intuitive method associates a state variable with every

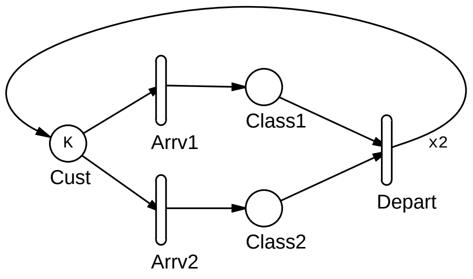

Fig. 2
SPN representing an $M/M/1/K/K$ queue variation

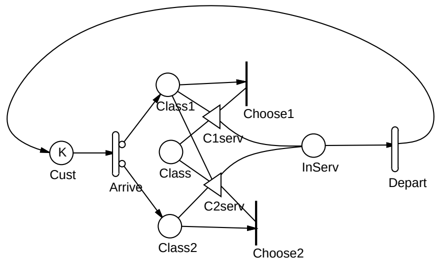

Fig. 3
A SAN variation of Figure 2

SPN place, and an action with every SPN transition. The formalism implementor would implement this by designing a class Place that inherits from BaseSV and a class Transition that inherits from BaseAction. This provides a framework for SPN models in general, but the classes must be used to represent *particular* SPN models as required by the modeler. Note that model-specific places differ from each other only in their names, markings, and enabling and affecting actions. Therefore, to create a Möbius representation of a particular model, the formalism designer need only ensure that objects of type Place are created for each model place, and instantiated with the correct data. Möbius insists on a fixed-size representation for each state, and an appropriate choice for an SPN place may be an `unsigned short`. Using that design, the methods **SetState** and **CurrentState** are easy to implement. In practice, class Place provides a method called Mark, implemented as a C++ *lvalue*, which can be used to copy or change the value of the state variable. The AFI for state variables also provides methods **AppendAffectingAction** and **AppendEnablingAction** to be used when initializing a model, for adding affecting and enabling actions.

Individual SPN transitions in a model differ in non-trivial ways, and the AFI supports implementation of these differences. For example, for two different transitions in any given SPN, the required implementations of **Enabled** will differ. For each specific SPN transition, the formalism designer may create a new class TransitionName that inherits from Transition and provides specific implementations of the key AFI methods. This is best illustrated by example:

• place **Cust** would be represented by a single object of type Place with initial value set to K. The AFI can be used to initialize **Cust** with data such that **AffectingActions** returns the list $\{\mathbf{Depart}\}$ and **EnablingActions** returns the list $\{\mathbf{Arrv1}, \mathbf{Arrv2}\}$.

• transition **Depart** would be represented by an object of type *Depart*, a class that inherits from Transition. For class *Depart*, AFI methods would be implemented as follows:

– **Enabled** would return true if both **Class1**.Mark() and **Class2**.Mark() evaluate to non-zero values.

– **Fire** would decrement both **Class1**.Mark() and **Class2**.Mark(), and increment **Cust**.Mark().

– **SampleDistribution** would return a `double` from an implementation of an exponential PDF with a parameter equal to the modeler's choice of parameter for **Depart**.

– **AffectedSVs** gives the list $\{\mathbf{Class1}, \mathbf{Class2}, \mathbf{Cust}\}$.

– **EnablingSVs** gives the list $\{\mathbf{Class1}, \mathbf{Class2}\}$.

3.1.2 Extensions to SPNs

Generalized SPNs (GSPNs) extend SPNs with the addition of immediate transitions, with which priorities and weights are associated. SANs are similar to GSPNs, but differ in the following ways:

• Activities (similar to transitions in GSPNs) may now have *cases*. If an activity with cases completes, then one case is chosen probabilistically according to a modeler-supplied distribution, and the marking will change according to the components connected to that case only.

• Activities may be connected to *input gates*, and input gates to places. An input gate specifies a modeler-supplied predicate that must be satisfied in that state, in order for connected activities to be enabled in the current state. Input gates also have modeler-specified functions that allow for arbitrary changes in the marking when connected activities complete. SANs also have *output gates*, which are not used in this example.

• An activity may have a *Reactivation predicate* and *Reactivation function*. These allow the modeler to express an execution policy on a per activity basis, as explained in Section 2.3.

Consider Figure 3, which extends our original model to use SAN components. Now, an arriving customer is probabilistically designated a class. Instantaneous activity $\mathbf{Choose}_i$ is enabled if place $\mathbf{Class}_i$ is marked and the predicate of gate $\mathbf{C}_i\mathbf{serv}$ is true, where the intention is to give priority to Class 1 customers. The predicate for gate $\mathbf{C_2serv}$ may be expressed in C++ as

```
(InServ->Mark() == 0) && (Class1->Mark() == 0)
```

Place **Class** holds a value of either 1 or 2, representing the class of a customer that is currently in service. This would allow activity **Depart** to have a completion rate dependent on the customer type being served. The formalism implementor need do little extra to capture this extra functionality. Immediate activities are captured in the AFI by Möbius actions in the same manner as for SPNs, and are implemented such that the respective **Weight** methods return the respective immediate activity priorities. Gate predicates can be smoothly incorporated using the **Enabled** method, and gate functions by using the **Fire** method. Probabilistic cases on activities can be captured by considering each (action,case) pair as an action itself, and placing all such pairs containing the same action in a group. A case may be selected by using post-selection on the group, as explained in Section 2.3.

```
#include "Cpp/BaseClasses/SAN/SANModel.h"
#include "Cpp/BaseClasses/SAN/Place.h"
extern short K;
extern double lambda;

class ExSanSAN : public SANModel {
public:
  class DepartActivity : public Activity {
  public:
    Place *InServ, *Class, *Cust;
    DepartActivity();
    bool Enabled();
    double SampleDistribution();
    BaseActionClass* Fire();
    bool ReactivationPredicate(); ...
  }

  Place *InServ, *Cust; ...
  DepartActivity Depart; ...
  PreselectGroup ImmediateGroup; ...
  ExSanSAN();
  ~ExSanSAN(){};
}
```

Fig. 4
C++ header file for model in Figure 3

3.1.3 The Implementation

The SAN formalism was the first Möbius atomic model formalism implemented. The model presented in Figure 3 was entered into the Möbius tool; then, as described in Section 2.1, C++ source code that inherits from the AFI base classes, and that represents the model, was automatically generated. Figure 4 shows the header file[1], including a class definition of SANModel, which inherits from BaseModel. Notice that a class is created for activity **Depart**, with declarations for each method that must be implemented. Each activity has available within its scope pointers to the state variables that its methods, such as **Enabled** and **Fire**, will need to access. Figure 5 shows parts of the generated C++ model source code. The constructor for the model base class declares objects to represent state variables, and then uses AFI actions to initialize the enabled and affected action data structures for the state variables. Figure 5 also shows the implementation for two AFI methods of SAN activity **Choose2**. Notice that the enabled condition is the conjunction of the presence of a token in place **Class2** and the satisfaction of the input gate predicate. In the generation of method **Enabled**, the code for the presence of the token is included automatically, and Möbius simply includes the gate predicate supplied by the modeler. For the **Fire** method, Möbius again uses a combination of a condition determined by the structure of the net, and a condition specified by the user as the input gate function.

```
#include "PNPM01/Atomic/ExSan/ExSanSAN.h"

ExSanSAN::ExSanSAN(){
  Cust = new Place("Cust" ,K);
  Cust−>appendAffectingAction(&Depart);
  Cust−>appendAffectingAction(&Arrive_1); ...
  Cust−>appendEnabledAction(&Arrive_1); ...
} ...

bool ExSanSAN::Choose2Activity::Enabled(){
  OldEnabled=NewEnabled;
  NewEnabled=((Class2−>Mark() > 0) &&
    ((InServ−>Mark() == 0) && (Class1−>Mark() == 0)));
                                          // from modeler
  return NewEnabled;
} ...

BaseActionClass* ExSanSAN::Choose2Activity::Fire(){
  InServ−>Mark()++;                       // from modeler
  Class−>Mark() = 2;                      // from modeler
  Class2−>Mark()−−;
  return this;
} ...
```

Fig. 5
C++ source code for model in Figure 3

Möbius uses an action group to hold all instantaneous SAN activities present in the model; on a state change, one is pre-selected to complete, based on rank and weight. Möbius also uses a group to hold an action with probabilistic cases, employing post-selection as described earlier in Section 3.1.2.

3.2 Realizing SPA-Style Languages

Möbius also supports an alternative style of modeling by providing PEPA (Performance Evaluation Process Algebra [12]), an SPA, as another atomic model formalism. Formally, PEPA models are specified as terms of a simple algebra. PEPA extends classical

[1]All code presented has been cut down manually for the sake of brevity.

process algebra with the capacity to assign rates to activities, leading to the definition of a stochastic process. PEPA has been applied to modeling the performance of distributed computer systems, and, for example, components of a flexible manufacturing system [13]. In contrast to the graph-based approach of Petri nets, building PEPA models is analogous to writing programs. More details on incorporating PEPA into Möbius can be found in [14].

A PEPA model $(\alpha, r).S$ may perform an action α at rate r and evolve into S; a model $S + T$ expresses a competition between S and T over actions, with the winner of the race determining the next model state. These *sequential components* are composed to produce *model components* that express the static structure of the model. $P \underset{L}{\bowtie} Q$ expresses the parallel composition of two submodels, with action synchronization on activity names in L.

In order to provide a useful mapping to the Möbius AFI, we enhanced PEPA with some convenient modeling features to produce PEPA$_k$, detailed in [14]. A simple example model illustrating these extensions is given in Figure 6. Parameter m represents the current number of customers in the queue, and n the maximum allowed. PEPA$_k$ exploits the well-known theory of process parameters, allowing the modeler to associate variables with sequential processes. These variables may then be used in *guards*; for example, in the model above, if the current model state is $Queue[0, 5]$, then the only activity enabled will be $(\mathtt{in}, \mu)$. Process parameters may also be communicated between activities, and the values altered after activities complete. However, despite this apparent increase in expressiveness, it can be shown that models expressed in PEPA$_k$ can be mechanically translated into PEPA models.

Process parameters are represented by state variables, and activities are represented by actions. The visible state variables of $Queue[m, n]$ are $\{m, n\}$. The state of the whole model is given by the values of the parameters and also by the current algebraic term. We take the approach of viewing the syntax of the model as a (static) tree of sequential components, and creating a state variable sv_S for each such "leaf" process S. For the model in Figure 6, the entire set of state variables is thus $\{m, n, sv_{Queue}\}$. For the term $Queue'[3, 5]$, the current values of the state variables would be $m = 3$, $n = 5$, and $sv_{Queue} = 1$, for some fixed indexing of the terms of the model.

$$
\begin{aligned}
Queue[m, n] &\overset{\text{def}}{=} [m > 0] \Rightarrow (\mathtt{out}, \lambda).Queue[m - 1, n] \\
&\quad + \; [m < n] \Rightarrow (\mathtt{in}, \mu).Queue'[m, n] \\
Queue'[m, n] &\overset{\text{def}}{=} (\mathtt{ref}, \rho).Queue[m + 1, n]
\end{aligned}
$$

Fig. 6
PEPA$_k$ model of a queue with customer registration

```
bool PEPAActivity::Enabled(){
  OldEnabled=NewEnabled;
  NewEnabled=(EnabledByCurrentTerms() && GuardSatisfied());
  return NewEnabled;
}
```

```
#include "PNPM01/Atomic/ExPEPA/ExPEPAPEPA.h"
. . .
inline bool ExPepaPEPA::out_Act::GuardSatisfied() {
  return (! (m->getValue() <= 0));
}

BaseActionClass *ExPepaPEPA::out_Act::Fire() {
  UpdateCurrentTerms();
  m->setValue((m->getValue() - 1));
  n->setValue(n->getValue());
  return this;
}
```

Fig. 7
AFI method and model-specific methods for Figure 6

The Möbius actions of a PEPA$_k$ model are given by the union of the types of all possible activities that the model could enable for any starting state[2]. For the model above, the Möbius actions would be given by the set $\{\mathtt{in}, \mathtt{out}, \mathtt{ref}\}$. If our model of interest was a cooperation between several components, then Möbius actions would be created for each possible synchronization of PEPA activities. The interested reader is referred to [14] for more details.

To map PEPA to the Möbius AFI, we designed a class PEPAVariable that inherits from BaseSV and a class PEPAActivity that inherits from BaseAction. Figure 7 illustrates some of the differences from the SAN implementation. For a PEPA model, an activity's enabling condition depends only on the syntactical form of the current term, and on the result of evaluating any guards. We implemented **Enabled** in class PEPAActivity itself, and factored out the methods needed for the enabling condition such that they are provided by each PEPA model. Method EnabledByCurrentTerms makes use of a representation of the syntax of the PEPA model kept internal to Möbius. Figure 7 shows method GuardSatisfied and the AFI method **Fire** for activity $(\mathtt{out}, \lambda)$. **Fire** updates the internal representation of the term, and then changes the values of the state variables appropriately.

3.3 Realizing Composed Models

Möbius also allows the construction of *composed models* from previously defined models. This allows the modeler to adopt a hierarchical approach to modeling, by constructing submodels as meaningful units

[2]If the model features two different copies of an activity of type a, they are distinguished as Möbius actions.

and then placing them together in a well-defined manner to construct a model of a system. One composition method in Möbius is the *state-sharing approach*; in this approach, submodels are linked together by identifying sets of state variables. For example, two Petri net models may be composed by holding a particular place in common. This allows for interaction between the submodels, since both can read from and write to the identified state variable. This form of state sharing is known as *equivalence sharing*, since both submodels have the same relationship to the shared state variable.

The AFI supports state sharing among submodels. The composed model presents the abstract functional interface to Möbius, and relies on AFI method calls to its submodels in order to realize the composition. That means that Möbius models are closed under composition. Currently, Möbius features two composed model formalisms.

3.3.1 Replicate/Join

The *Replicate/Join* composed model formalism was originally conceived for SAN models (see [15]). It enables the modeler to define a composed model in the form of a tree, in which each leaf node is a predefined atomic or composed model, and each non-leaf node is classified as either a *Join* node or a *Replicate* node.

A Join is used to compose two or more submodels using equivalence sharing. A Join node may have further Joins, Replicates, or other models defined as its children. The state of a Join composed model is given by a vector of the states of each of its children.

A Replicate is used to construct a model consisting of a number of identical copies of a submodel. The resulting composed model is equivalent in behavior to that which would result from a Join composed model in which all the components are copies of the same submodel. A Replicate node has one child, which may be another Replicate, a Join, or a single atomic or composed model. The modeler may also specify a set of state variables to be held in common between all replicated instances of the submodel.

Since the instances of a Replicate composed model are indistinguishable, its state can be represented in a lumped way, as a sequence of non-negative numbers $n_1, n_2, \cdots, n_N$ such that $\sum_{i=1}^{N} n_i = m$ where N is the number of states of an instance, m is the number of instances, and n_i is the number of instances in the ith state. The sequence shows how many of the instances are in each state. This optimization is known as *reduced base model construction*, and a more detailed description is available in [15]. Since the composed model presents its state to Möbius through the AFI, it can keep details of symmetry-based reductions private. The rest of the Möbius tool does not and has no need to know the details of such optimizations.

3.3.2 Graph Composition

Möbius supports a second composed model formalism called *Graph Composition* [16,17]. As with Replicate/Join, the fundamental operation is the joining of two or more models through equivalence sharing. However, whereas the structure of a Replicate/Join composed model is a tree, the structure of a Graph composed model is a graph, where an arc linking two models indicates that the two models exhibit an equivalence-sharing relationship.

As with Replicate/Join composition, lumping techniques can be used to automatically find all symmetries present in the graph structure of the model [16]. For Graph composed models, these symmetries are more varied; for example, the model may be invariant under a rotation of its graph. The methods to detect these symmetries are based on computational group theory, and, when symmetry is present, can drastically reduce the size of the state space of the composed model. The current implementation of graph composition in Möbius supports this formalism, and work is underway to implement the lumping techniques. As with Replicate/Join, a Graph composed model presents the abstract functional interface to the rest of Möbius, and relies on AFI calls to keep track of the state of its submodels as far as is required by model symmetries.

4 Building Solvable Models

Reward models [18] build upon atomic and composed models, equipping them with the specification of a performance measure. At this time we have implemented one type of reward model in Möbius: a *performance variable* (PV). A PV[3] allows for the specification of a measure on one or both of the following:

- the states of the model, giving a *rate reward* PV
- action completions, giving an *impulse reward* PV

A rate reward is a function of the state of the system at an instant of time. An impulse reward is a function of the state of the system and the identity of an action that completes, and is evaluated when a particular action completes. A performance variable can be specified to be measured at an instant of time, measured in steady state, accumulated over a period of time, or accumulated over a time-averaged period of time. Once the rate and impulse rewards are defined, the desired statistics on the measure must be specified. The options include solving for the mean, variance, or distribution of the measure, or the probability of the measure falling within a specified range.

[3]Note that although these variables are called *performance variables*, they are generic and can be used to represent dependability and performability variables as well.

During the specification of atomic, composed, and reward models, *global variables* can be used to parameterize model characteristics. A global variable is a variable that is used in one or more models, but not given a value. In the Möbius tool, global variables can have any of the basic C++ types, including `short`, `int`, and `double`. Examples are shown in Figure 4; specifically, K and lambda are defined to be global variables in the SAN model of Figure 3. Möbius implements this functionality by declaring these variables as `extern`. This allows their values to be defined in a separately compiled unit.

Models are solved after each global variable is assigned a specific value. One such assignment forms an *experiment*. Experiments can be grouped together to form a *study*. Möbius supports several study editors, the most sophisticated of which is based on a Design of Experiments approach (DOE [19]). A DOE study generates a set of experiments, and then analyzes the reward variable solutions to determine how the chosen global variables affect the reward variables. Sensitivity analysis can measure the effects of all model parameters and their interactions on each solved reward variable. In addition, the model parameter values that produce optimal reward variable values can be determined.

```
TRANSIENT SIMULATOR()
$S \leftarrow Model.$CurrentState()
for ($Act \in Actions$) do
    if ($Act.$Enabled()) then
        $Enabled \leftarrow Enabled \cup \{Act\}$
while (! $SimServ\ Converged$) do
    $Time \leftarrow 0.0$
    $EvtList \leftarrow SimCli.$GenEventList($Enabled$)
    $Evt \leftarrow EvtList.$Earliest()
    while ($Time < StopTime$) do
        $Time \leftarrow Time + Event.firetime$
        $Evt.Act.$Fire()
        for $\{Act : SV \in Evt.Act.$AffectedSVs()
                $and\ SV \in Act.$EnablingSVs()
                $and\ Act \in EvtList\}$ do
            if ($Act.$EnablingChange() $or$
                    ($Act.$Reactivation $and$
                    $Act.$ReactivationFunction())) then
                $Act.$Reactivation $\leftarrow false$
                $EvtList.$Remove($Act$)
        for $\{Act : SV \in Evt.Act.$AffectedSVs()
                $and\ SV \in Act.$EnablingSVs()
                $and\ Act.$Enabled()
                $and\ Act \notin EvtList\}$ do
            $firetime \leftarrow Act.$SampleDistribution()
            $EvtList.$Add(new $Evt(firetime)$)
    $SimCli\ reports\ results$
return $true$
```

Fig. 8
PSEUDO-CODE FOR THE MÖBIUS TRANSIENT SIMULATOR

5 Solving Models

The Möbius tool currently supports two classes of solution techniques: discrete event simulation and state-based, analytical/numerical techniques. Any model specified using Möbius may be solved using simulation. Models that have delays that are exponentially distributed, or have no more than one concurrently enabled deterministic delay, may be solved using a variety of analytic techniques applied to a generated state space. The simulator and state-space generator operates on models only through the Möbius AFI.

5.1 Simulation

The Möbius tool currently has two discrete event simulators: a transient simulator and a steady-state simulator. The transient simulator uses the independent replication technique to obtain statistical information about the specified reward variables. The Möbius steady-state simulator uses batch means with deletion of an initial transient to solve for steady-state, instant-of-time variables. Estimates available during simulation include mean, variance, interval, and distributions, and the results for mean and variance provide confidence intervals.

Both Möbius simulators may be executed on a single workstation, or distributed on a network of workstations. The network may be a mixture of any supported architectures or operating systems. This parallelism is accomplished by running different observations on different workstations in the case of transient simulation, or different trajectories in the case of batch means. We have observed that this level of parallelism usually yields near-linear speedup.

Figure 8 presents (simplified) pseudo-code for the operation of the Möbius transient simulator. Through the AFI, the simulator makes use of the construction of affects and enables lists to simulate efficiently the execution of the model.

5.2 State-Space Generator

Möbius also supports a variety of analytical/numerical solvers. The first step in analytic solution with Möbius is the generation of a state space, done by the Möbius state-space generator. Note that symmetries in the model are detected and leveraged by the various composition formalisms, and since the state-space generator only accesses the model through the AFI, it need not and does not know the details of these reductions. Furthermore, the state-space generator may be employed on any Möbius model. This allows the state-space generator to be generic, so it need not understand the semantics of a model on which it is operating. Once the state space is generated, any

```
State-Space Generator()
    $AllStates \leftarrow \emptyset$
    $States \leftarrow \{Model.\mathsf{CurrentState}()\}$
    $Actions \leftarrow Model.\mathsf{ListActions}()$
    while ($States \neq \emptyset$) do
        $S \leftarrow States.\mathsf{Pop}()$
        $Model.\mathsf{SetState}(S)$
        $Enabled \leftarrow \emptyset$
        for ($Act \in Actions$) do
            if ($Act.\mathsf{Enabled}()$) then
                $Enabled \leftarrow Enabled \cup \{Act\}$
        while ($Enabled \neq \emptyset$) do
            $Act \leftarrow Enabled.\mathsf{Pop}()$
            $Act.\mathsf{Fire}()$
            $S' \leftarrow Model.\mathsf{CurrentState}()$
            if ($S' \notin AllStates$) then
                $States \leftarrow States \cup \{S'\}$
                $AllStates \leftarrow AllStates \cup \{S'\}$
            $Model.\mathsf{SetState}(S)$
    return $AllStates$
```

Fig. 9
Pseudo-code for the Möbius state-space generator

of several implemented analytical/numerical methods may be employed to solve for the required performance variables.

Figure 9 presents pseudo-code for the operation of the Möbius state-space generator.

6 Conclusion

Through careful definition of a modeling framework and the AFI, we were able to construct a modeling tool that is extensible and supports multiple modeling formalisms and model solution methods. To date, we have implemented three atomic model formalisms (SANs, buckets and balls (an extended Markov chain formalism [20]), and PEPA), two composition formalisms (Replicate/Join and Graph composition), one reward variable specification formalism, and several study editors. Multiple simulation and analytical/numerical solvers have also been implemented. Additional modeling formalisms and solution methods are currently under development by others and ourselves. This list gives evidence for our claim that the AFI makes Möbius extensible. These successes bode well for Möbius's use as a vehicle for others to implement new modeling formalisms and solution methods, and we welcome participation by others in this endeavor.

ACKNOWLEDGMENTS

We would like to acknowledge the contributions of the former members of the Möbius group: Amy L. Christensen, G. P. Kavanaugh, John M. Sowder, Aaron J. Stillman, and Alex L. Williamson.

REFERENCES

[1] W. H. Sanders, "Integrated frameworks for multi-level and multi-formalism modeling," in *Proceedings of PNPM'99: 8th International Workshop on Petri Nets and Performance Models, Zaragoza, Spain*, September 1999, pp. 2–9.

[2] R. A. Sahner and K. S. Trivedi, "Reliability modeling using SHARPE," *IEEE Transactions on Reliability*, vol. R-36, no. 2, pp. 186–193, June 1987.

[3] G. Ciardo and A. S. Miner, "SMART: Simulation and markovian analyzer for reliability and timing," in *Proceedings of IEEE International Computer Performance and Dependability Symposium (IPDS'96)*, September 1996, p. 60.

[4] F. Bause, P. Buchholz, and P. Kemper, "QPN-Tool for the specification and analysis of hierarchically combined queueing Petri nets," *Lecture Notes in Computer Science*, vol. 977, pp. 224–238, 1995.

[5] J. M. Doyle, "Abstract model specification using the Möbius modeling tool," M.S. thesis, University of Illinois at Urbana-Champaign, Jan. 2000.

[6] W. H. Sanders, W. D. Obal II, M. A. Qureshi, and F. K. Widjanarko, "The *UltraSAN* modeling environment," *Performance Evaluation*, vol. 24, no. 1, pp. 89–115, October-November 1995.

[7] J. F. Meyer, A. Movaghar, and W. H. Sanders, "Stochastic activity networks: structure, behavior and applications," *Proc. International Workshop on Timed Petri Nets*, pp. 106–115, 1985.

[8] B. Plateau and K. Atif, "A methodology for solving Markov models of parallel systems," *IEEE Journal on Software Engineering*, vol. 17, no. 10, pp. 1093–1108, 1991.

[9] D. Deavours and W. H. Sanders, "Möbius: Framework and atomic models," in *PNPM'01: 10th International Workshop on Petri Nets and Performance Models, Aachen, Germany (to appear)*, September 2001.

[10] M. Ajmone Marsan, G. Balbo, A. Bobbio, G. Chiola, G. Conte, and A. Cumani, "The effect of execution policies on the semantics and analysis of stochastic Petri nets," *IEEE Transactions on Software Engineering*, vol. 15, no. 7, pp. 832–846, July 1989.

[11] D. Deavours and W. H. Sanders, "The Möbius execution policy," in *PNPM'01: 10th International Workshop on Petri Nets and Performance Models, Aachen, Germany (to appear)*, September 2001.

[12] J. Hillston, *A Compositional Approach to Performance Modelling*, Cambridge University Press, 1996.

[13] D. R. W. Holton, "A PEPA specification of an industrial production cell," *The Computer Journal*, vol. 38, no. 7, pp. 542–551, 1995.

[14] G. Clark and W. H. Sanders, "Implementing a stochastic process algebra within the Möbius modeling framework," in *PAPM'01: 9th International Workshop on Process Algebra and Performance Models, Aachen, Germany (to appear)*, September 2001.

[15] W. H. Sanders and J. F. Meyer, "Reduced base model construction methods for stochastic activity networks," *IEEE Journal on Selected Areas in Communications*, vol. 9, no. 1, pp. 25–36, Jan. 1991.

[16] W. D. Obal II, *Measure-Adaptive State-Space Construction Methods*, Ph.D. thesis, The University of Arizona, 1998.

[17] A. J. Stillman, "Model composition within the möbius modeling framework," M.S. thesis, University of Illinois at Urbana-Champaign, 1999.

[18] W. H. Sanders and J. F. Meyer, "A unified approach for specifying measures of performance, dependability, and performability," in *Dependable Computing for Critical Applications*, A. Avizienis, J. Kopetz, and J. Laprie, Eds. 1991, vol. 4 of *Dependable Computing and Fault-Tolerant Systems*, pp. 215–237, Springer-Verlag.

[19] D. Montgomery, *Design and Analysis of Experiments*, John Wiley & Sons, Inc., 5th edition, 2001.

[20] W. J. Stewart, *Introduction to the Numerical Solution of Markov Chains*, Princeton University Press, 1994.