Möbius: Framework and Atomic Models *

Daniel D. Deavours and William H. Sanders

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.
E-mail: {deavours,whs}@crhc.uiuc.edu

Abstract

This paper gives an overview of the Möbius framework, and gives a formal specification for defining atomic models within the framework. The framework is designed to be capable of incorporating multiple modeling formalisms, including atomic models (e.g, SPNs), composition formalisms (e.g., Replicate/Join), measure specification formalisms, connection formalisms, and solvers. We focus on atomic models, which are composed of actions, state variables, and properties. We argue that these are sufficient to specify a large number of atomic model formalisms in the Möbius framework. The framework serves as a basis for the Möbius tool [13].

1 Introduction

Performance and dependability modeling is an important part of the design process of many computer and communication systems. A variety of techniques have been developed to address different issues of modeling. For example, combinatorial models were developed to assess reliability and availability under strong independence assumptions; queueing networks were developed to assess system performance; and Markov process-based approaches have become popular for evaluating performance with synchronization or dependability without independence assumptions. Finally, simulation has been used extensively when other methods fail.

As techniques for solving models advanced, the formalisms used for expressing models were also developed. Each formalism has its own merits. Some formalisms afford very efficient solution methods; for example, BCMP [2] queueing networks admit product-form solutions, while superposed generalized stochastic Petri nets (SGSPNs) [19] afford Kronecker-based solution methods, and colored generalized stochastic Petri nets (CGSPNs) [6] yield state-space reductions. Other formalisms, such as SPNs [26], provide a simple elegance in their modeling primitives, while a number of extensions, such as stochastic activity networks (SANs) [25], were developed for compactly expressing complex behaviors.

Along with formalisms, tools have been developed. A tool is generally built around a single formalism and one or more solution techniques, with simulation sometimes available as a second solution method. [32] lists a number of tools that fall into this category, such as DyQN-Tool+ [22], which uses dynamic queuing networks as its high-level formalism, GreatSPN [7], which is based on generalized stochastic Petri nets (GSPNs) [1], *UltraSAN* [35], which is based on SANs [25], SPNP [11], based on stochastic reward networks [8], and TANGRAM-II [5], which is an object- and message-based formalism for evaluating computer and communication systems. While all of these tools are useful within the domain for which they were intended, they are limited in that all parts of a model must be built in the single formalism that is supported by the tool.

1.1 Related work

A more complete discussion of related work can be found in [32]; we briefly review tools most closely related to Möbius here.

One approach has been what we call the “integrated software environment” approach. This approach seeks to unify several different modeling tools like the ones described above into a single software environment. Examples are IMSE (Integrated Modeling Support Environment) [30], IDEAS (Integrated Design Environment for Assessment of Computer Systems and Communication Networks) [21], and Freud [37]. One problem with these approaches is that since they use existing tools, the degree to which formalisms and solution methods of different tools may interact is limited.

* 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). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or of Motorola.

A more aggressive approach is the "multi-formalism multi-solution" approach, in which a tool implements more than one formalism and solution technique. Models expressed in different formalisms may interact by passing results from one model to another. The earliest attempt to do this, to the best of our knowledge, was the combination of multiple modeling formalisms in SHARPE [31]. In the SHARPE modeling framework, models can be expressed as combinatorial reliability models, directed acyclic task precedence graphs, Markov and semi-Markov models, product-form queueing networks, and generalized SPNs. Interactions between formalisms are limited to the exchange of results, either as single numbers or as exponential-polynomial probability distribution functions. Another tool that integrates multiple modeling formalisms in a single software environment is SMART [9]. SMART supports the analysis of models expressed as SPNs and queueing networks, and the tool is implemented in a way that permits the easy integration of new solution algorithms. The DEDS (Discrete Event Dynamic System) toolbox [3] also integrates multiple modeling formalisms into a single software environment, but does so by converting models expressed in different modeling formalisms into a common "abstract Petri net notation." Once expressed in this abstract formalism, models may be solved using a variety of functional and quantitative analysis approaches for Markovian models.

1.2 The Möbius approach

The approach we take with Möbius [32] is an integrated multi-formalism multi-solution approach. Our goal is to build a tool in which each model formalism and solver is, to the extent possible, modular, in order to maximize potential interaction. This goal is possible because many operations on models, such as composition (described later), state-space generation, and simulation are largely independent of the formalism being used to express the model.

This approach has several advantages. First, it allows for novel combinations of modeling techniques. For example, to the best of our knowledge, the Replicate/Join model composition approach of [33] has been used exclusively with SANs. This exclusivity is artificial, and in the Möbius tool, Replicate/Join can be used with virtually any formalism that can produce a Markov chain, such as PEPA [12].

Another advantage is the ease with which new approaches may be integrated into the tool. Perhaps the most convincing argument for this comes from our own experience. To the extent possible, we have incorporated new research results into our successful modeling tool, *UltraSAN*. For a number of practical reasons, many of our recent research results have only been developed to a prototype, and are not generally available to other users (for example, [15, 16, 17, 27, 28]). In particular, we would have liked to develop a new graph composition formalism of [28], but found it difficult to include in *UltraSAN* because of the inherently closed nature of the software design. Möbius has been designed to include these modeling techniques, as well as to be extensible to include future techniques.

The ability to simply add new components would benefit researchers and users alike. Researchers would be able to add new components to the tool and have that component immediately interact with other components. They would also be able to compare competing techniques directly. Researchers would have access to the work of others, and be able to extend and compare techniques. Users would also benefit by having access to the most recent developments in conjunction with previously existing techniques. Having a modular, "toolbox" approach would be valuable for users, allowing the user to choose the most appropriate tool or tools for the job.

While we would like Möbius to have a great deal of flexibility, we would also like to be able to retain the ability to perform efficient solution. Efficient solution is usually possible because of some special structure or condition that is met in the model. Wherever possible, we would like Möbius to retain the ability to perform efficient solution.

For all these reasons, we argue that an open, multi-formalism, multi-solution modeling framework and tool would represent a significant step forward in advancing the state-of-the-art in performance/dependability modeling techniques and tools. The simple motivating concept behind this is that a framework allows for maximum potential interaction of techniques. As much as possible, formalisms, ways of combining models, measure specification, and solution should be independent operations. The price for this interaction is the need to create a sufficiently general and abstract representation of a model, while at the same time retaining the ability to perform efficient solutions. While this is not trivial, we believe the benefits are worth the effort. To this end, we have created the Möbius framework.

1.3 Outline

We present an overview of the Möbius framework in Section 2. The remainder of the paper then focuses on one important part of the framework: the atomic model. An atomic model consists of state variables, actions, and properties, which are discussed in Sections 3, 4, and 5.1 respectively. We conclude in Section 7 with a brief description of a tool based on this framework.

A framework is a substantial undertaking, and for any particular choice we made in defining what is within the framework, and equally, what is outside of the framework, it is legitimate to question the choice. We do not claim that our choice is in some sense "complete," nor do we argue that we have always made the "right" choices by proof. We simply offer evidence (often in the form of examples) that we studied many formalisms, and that most of them seem to fit nicely within this framework. We believe that this is the best that can be done, given that our measures of success include modeling convenience and the ability to facilitate efficient and appropriate interactions between model

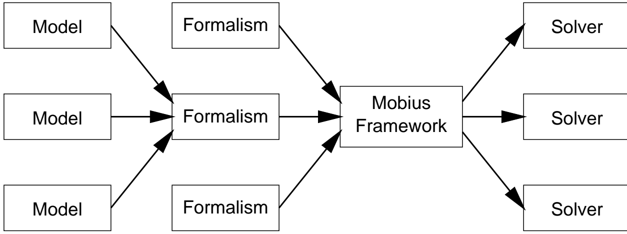

Figure 1. Models, formalisms, and framework.

components. Also, due to space limitations, we can not reference all the relevant work, so we have chosen to present those references with which we are most familiar.

2 Möbius framework

The Möbius framework is the formal specification of an environment in which multiple modeling formalisms and solvers can interact. As we stated earlier, the goal of the Möbius framework is to maximize the amount of interaction possible between formalisms and solvers.

2.1 Framework description

Models are expressed in particular formalisms within the Möbius framework. A modeling formalism, in order to be compatible with the framework, must represent the various model components as framework components. In this way, models of different formalisms and solvers can interact because they are interacting with framework components.

Figure 1 illustrates the relationship between models, formalisms, and solvers within the framework. Users of the Möbius tool enter models in a particular modeling formalism. The formalism editor, in turn, translates each component of the model into a corresponding component of the framework. The user is able to utilize all the advantages of a particular formalism, while at the same time having access to all the other components of the tool, such as various composition, measure, and connection methods, as well as a variety of solvers. This also allows the tool to be extensible, because we can add new formalisms and solvers with a minimum of impact on existing formalisms and solvers.

In order to accomplish that, framework components must be general enough to express a variety of different formalism components. Note a subtle but important point concerning the Möbius framework: we are not trying to develop a universal formalism. While formalisms may express only a subset of what is possible within the framework, we believe that the subsets expressed by formalisms are carefully chosen by researchers with years of experience to accomplish various purposes. Although its generality is important, we believe that the real value of the framework lies in its ability to accept the addition of new formalisms and solvers in a modular way. There may be some value in developing a formalism that maps directly to the framework and thereby exposes the full generality that is possible within the framework, but that is not the focus of this paper.

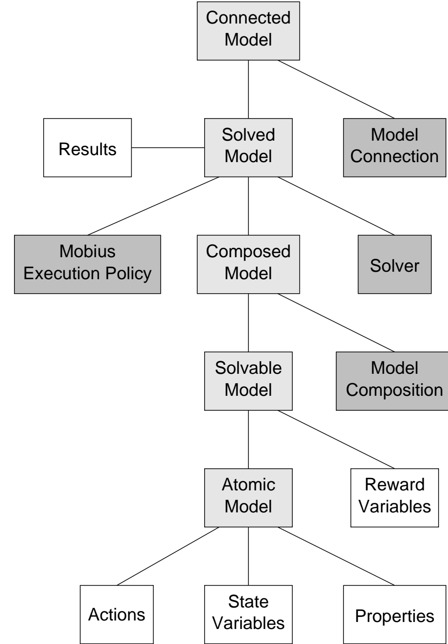

Figure 2. Möbius framework components.

2.2 Framework components

In order to define the framework, we must abstract away many of the concepts found in most formalisms. We also must generalize the process of building and categorizing models. We begin with an illustration shown in Figure 2, which outlines the various components within the Möbius framework.

The first step in the model construction process is to generate a model or submodel using some formalism. This most basic model is called an *atomic model*, and is made up of state variables, actions, and properties. State variables (for example, places in the various stochastic extensions to Petri nets (PN), or queues in queueing networks) hold state information about a model, while actions (such as transitions in SPNs or servers in queueing networks) are the mechanism for changing model state. Properties provide information about a model that may be required in order to use a specialized solver, or to make the solution process more efficient for some solvers. The atomic model is the topic of the remaining sections of this paper.

After a model or submodel is created, frequently the next step is to specify some measures of interest on the model using some reward specification formalism, e.g., [34]. The Möbius framework captures this pattern by having a separate model type, called *solvable models*, that augments

atomic models with reward variables. Some formalisms may have the measure specification as part of the formalism description. In that case, the formalism produces a solvable model instead of an atomic model.

If the model being constructed is intended to be part of a larger model, then the next step is to *compose* it with other models to form a larger model. This is sometimes used as a convenient technique to make the model modular and easier to construct; at other times, the ways that models are composed can lead to efficiencies in the solution process. Examples of this include the Replicate/Join composition formalism [33] and the graph composition formalism of [28], in which symmetries may be detected and state lumping may be performed on the fly. Both of those formalisms have been implemented in the Möbius tool [13]. Other composed model techniques include synchronization on actions, which is found, for example, in stochastic process algebras (SPAs) such as PEPA [23], and well as in stochastic automata networks (also SANs, e.g., [29]) and superposed GSPNs (e.g., [19, 24]). Note that the compositional techniques do not depend on the particular formalism of the atomic models that are being composed, provided that any necessary requirements are met.

Model composition may preserve or destroy properties of the submodels (properties will be defined formally later), and it may add new properties to the resulting composed model. In our framework, solvable models may be composed, and the result of model composition is a composed model. Note that since composition occurs on solvable models, the composed model is also solvable. Furthermore, it is possible to add reward variables to a composed model, provided that they comply with any model compositional properties.

The next step is typically to apply some solver to compute a solution. We call any mechanism that calculates the solution to reward variables a *solver*. The calculation can be exact, approximate, or statistical. It may take advantage of model properties due to the atomic model formalisms and/or model composition. Note that solvers operate on framework components, not formalism components. Consequently, a solver may operate on a model independent of the formalism in which the model was constructed, so long as the model has the properties necessary for the solver.

The computed solution to a reward variable is called a *result*. Since the reward variable is a random variable, the result is expressed as some characteristic of a random variable. This may be, for example, the mean, variance, or distribution of the reward variable. The result may also include any solver-specific information that relates to the solution, such as any errors, the stopping criterion used, or the confidence interval. A solution calculated in this way may be the final desired measure, or it may be an intermediate step in further computation. If a result is intended for further computation, then the result may capture the interaction between multiple solvable models that together form a connected model.

A *connected model* is a set of solvable models in which input parameters to some of the models depend on the results of other models in the set. This is useful for modeling using decompositional approaches, such as that used in [10]. In those cases, the model of interest is a set of solvable models with dependencies through results, where the overall model may be solved through a system of nonlinear equations (if a solution exists).

We believe that the majority of modeling techniques can be supported within this framework. By making different modeling processes (such as adding measures, composing, solving, and connecting) modular, we can maximize the amount of interaction allowable between these processes. This also allows the tool to be extensible, in that new atomic modeling formalisms, reward formalisms, compositional formalisms, solvers, and connection formalisms may be added independently.

We now describe the basic type of Möbius model, the atomic model, which is made up of state variables, actions, and properties. We begin by describing state variables.

3 State variables

A state variable typically represents some component or subcomponent state. It may be as simple as the number of jobs waiting in a queue, or as complex as the state of an ATM switch.

Different formalisms represent state variables differently. For example, SPNs and extensions have places that contain tokens, so the values that a place can take on is the set of natural numbers. Colored GSPNs (CGSPNs) [6] have been extended so that tokens can take on a number of different colors at a place, making the value of a colored place a bag or multi-set. Queueing networks with different customer classes can have more complicated notions of state, such as those found in extended queueing networks [36], in which each job (customer) may have an associated job variable, which is typically implemented as an array of real numbers.

One goal in developing the Möbius framework is to capture and express all state variable types in existing formalisms so that it is possible to implement these various formalisms within the framework. We also have the goal of implementing new and interesting formalisms, possibly with more exotic state variable representations.

To do that, we must create a framework in which we can create general types of state variables. By using a framework, we enjoy all the benefits of a framework we discussed earlier. Specifically, solvers can interact with Möbius state variables, instead of with the variety of different formalism state variables. Composed model formalisms also need to interact only with Möbius state variables or actions. Finally, any efficiencies that may be gained through any structural knowledge can be preserved through the use of properties.

1. $\mathbb{Z} \in T$
2. If $i_1, i_2 \in \mathbb{Z} \cup \{\pm\infty\}$, $i_1 < i_2$, then $\{i_1, \ldots, i_2\} \in T$.
3. $\mathbb{R} \in T$
4. If $a, b \in \mathbb{R}$, $a < b$, then $([a, b)] \in T$.
5. $S \cup \{\nu\} \in T$
6. If $t \in T$, then $2^t \in T$.
7. If $t_1, t_2, \ldots, t_n \in T$, then $t_1 \times t_2 \times \cdots t_n \in T$
8. If $t_1, t_2, \ldots \in T$, then $t_1 \times t_2 \times \cdots \in T$.
9. If $t_1, t_2, \ldots, t_n \in T$ are disjoint, then $\cup_{i=1}^{n} t_i \in T$.

Figure 3. Construction of the type set T.

3.1 Components of State Variables

In the Möbius framework, a state variable is made up of three components: a value, a type, and an initial value distribution, which includes a set of possible initial states. The value of a component represents a particular configuration that the modeled component may be in, while the type of a state variable represents the range of values that the variable may take. The initial value distribution probabilistically gives the distribution of values at time 0. A distribution of values allows us to consider models whose initial states may vary. For example, it allows us to consider a system that is in steady state and measure the expected time until some important event.

3.1.1 Types

We begin by describing state variable types. Let $S = \{s_i\}$ be a set of state variables. (One can think of this as a set of state variable names.) Let T be the set of types that a state variable may take on, and let $type : S \rightarrow T$ be the type function. We construct T as the smallest set satisfying the rules shown in Figure 3. Here, $\mathbb{Z}$ is the set of integers, $\mathbb{R}$ is the set of reals, 2^t is the power set of t, and ν is a `nil` element. The meaning of $([a, b)] \in T$ is $[a, b], (a, b], [a, b), (a, b) \in T$.

We briefly describe some implications of the rules constructing T. "Integers" and "a subset of integers" are types; "reals" and "an interval over the reals" are types; pointers to state variables (including `nil`) are types; a set of a type is a type (e.g., set of integers); finite and infinite tuples are types (e.g., arrays or structures); finite unions of disjoint types are a type. Note that types of state variables are static and do not change throughout the evolution of a model.

3.1.2 Values

Naturally, state variables take on values, and the values of state variables change over time. As described informally earlier, the values that a state variable may take on are determined by its type. For example, if $type(s) = \mathbb{N}$, then the set of values that s may take on is $\mathbb{N}$. The value of a state variable is given by the value function $val : S \rightarrow V$ such that $val(s) \in type(s)$, where $V \in T$. The value function is not a formal element of the Möbius model, but is used in describing the execution of a model.

We say that the state of a state variable s is the pair $(val(s), type(s))$. We say that two state variables are *equal* if the states of the two state variables are equal. Thus, two state variables must have the same type and the same value to be equal. For example, the empty set of pointers is not equal to the empty set of integers, even though their values are equal (the empty set).

3.1.3 Initial value

It is convenient to be able to describe the initial value probabilistically. For example, the system may be described in steady state, during which it can be in a number of different states with various probabilities. To capture this behavior, we include a set of possible initial states and a probability function describing the probability of being in each initial state. Let $IS = \{val\}$ be a countable set of initial values for the state variables. Let $P_S : IS \rightarrow [0, 1]$ be the initial state probability distribution so that $P_S(v) = \Pr[\text{Initial value of state variables is } v]$. This allows the initial state to be probabilistically distributed among different values.

3.2 Properties

Properties are a set of symbols that specify that a certain condition or conditions about a state variable are true. They are intended to be used by specialized solvers that are applicable if certain conditions hold, or by solvers that take advantage of the information for more efficient solution.

Properties are nothing more than a set of symbols that have proprietary meaning. We write properties as <property>. Let Π be the set of all properties. The *state variable properties* is a function $SVP : S \rightarrow 2^{\Pi}$. An example property is <member P-invariant 1> $\in SVP(s_i)$, which may indicate that the state variable is part of a particular P-invariant; a state-space generator may take advantage of this by eliminating the need to explicitly store the value of one state variable in each P-invariant. Another example is <unsharable> $\in SVP(s_i)$, which may indicate that the state variable s_i may not be shared with a state variable of another model via model composition, perhaps because it is replicated.

Note that a property does not provide additional information about a model. Rather, it provides information about a condition that is true, but may be difficult or expensive to determine. For example, one could test each action of a model to determine whether it is exponentially distributed in all states, but that may be expensive compared to the effort required if a property already states that this is true.

3.3 Definition

Now we can formally define the state variables of a model to be the components

$$SV = (S, type, IS, P_S, SVP)\,.$$

3.4 Examples

We illustrate the usefulness and richness of state variable types with several examples.

A GSPN [1] place has type $\mathbb{N}$, which is a subset of $\mathbb{Z}$, so the type of a GSPN place can be formed using rule 2.

The value of a CGSPN [6] place can be expressed using a set of pairs: a color and the numbers of tokens of that color. Let C be a set of integers that enumerate the set of colors of place s. Note that $C \in T$ by rule 2. The number of tokens of a color in s is $\mathbb{N} \in T$ as shown above. Thus, the state of a colored place can be expressed as $C \times \mathbb{N} \in T$ by rule 7, where the first term represents the color, and the second term represents the cardinality.

SPAs are significantly different from SPN models because state variables are not as explicitly expressed in SPAs. For example, a PEPA [23] model state comprises the states of all the sequential components. Each sequential component may be represented as a state variable within the Möbius framework. Let some sequential component s have K stages. Each stage can be numbered $1, \ldots, K$. Note that $\{1, \ldots, K\} \in T$ by rule 2. We also note that discovering all the sequential components of a PEPA model involves some work, but this work is normally required for analysis, so it is a reasonable requirement.

Consider a finite FCFS queue with customer classes, where service time differs depending on the class [2]. Let C be the set of integers that enumerate the set of customer classes. We know $t_1 = \{0\} \cup C \in T$. Then let t_2 be the k-tuple $t_1 \times t_1 \times \cdots \times t_1$. $t_2 \in T$ by rule 7. If the queue is an infinite queue, then $t_3 = t_1 \times t_1 \times \cdots \in T$ by rule 8.

Next consider a finite priority queue with a preemptive resume policy. An action (discussed in Section 4) stores the age information for the active job of each priority. The state variable representing the queue can be the same one we constructed for the colored GSPN place.

Finally, we illustrate the extended queueing network (EQN) [36] job variable. A job variable is an array of k real numbers that is associated with each job. Using rule 7, we know $t_1 = \mathbb{R} \times \cdots \times \mathbb{R} \in T$ (a k-tuple). Let s_1 have type t_1 and represent a job. Again, using rule 2, we know $C \in T$, where C is the set of integers that enumerate the customer classes (colors). Using rules 5 and 7, we know $t_2 = S \cup \{\nu\} \times C \in T$. Using rule 7 or 8 (depending on whether the queue is finite or infinite), an array of t_2 is a valid type; call this new type t_3. Type t_3 can thus represent the EQN queue. If a customer is in a stage in the queue, then the customer class is indicated and the pointer (or reference) is set to the appropriate job variable. If no customer is in a stage in the queue, then the pointer (reference) is set to ν (nil). Notice that there are several different ways one could construct this in the Möbius framework. We chose only one for illustration.

4 Actions

An action represents the basic unit of a model that facilitates the changing of the state of state variables. An action corresponds to a transition in SPNs [26], GSPNs [1], and other extensions, an action of a SPA (e.g., [23]), an activity of a SAN [25], and a server of a queueing network (e.g., [2]), for example.

Möbius actions are similar to state variables in that their goal is to provide an abstraction of the various concepts of actions present in most formalisms. State-change mechanisms of atomic model formalisms in the Möbius framework may be implemented using a subset of the functionality provided by actions. Note that it is the restriction of the possible generality that often allows for efficiencies in solution methods. For example, restricting the delay times to be zero or exponential is useful because the underlying stochastic process is then Markovian. If behavior of a queueing formalism is restricted to "remove one job from one queue and add one job to another queue," along with several other properties, then a product form solution is possible.

An important aspect of the functionality of the action is what we call the *execution policy*. We use the term execution policy, as in [4], for a set of rules to unambiguously define the underlying stochastic process that describes the behavior of a model in the Möbius framework. We review describe the Möbius execution policy below.

Finally, like state variables, the Möbius action provides a common interface by which other model components (possibly of different formalisms) and solvers may interact. This allows for composition by synchronization, as is found in SPAs, stochastic automata networks (e.g., [29]), and superposed GSPNs (e.g., [19, 24]). These types of model composition methods are possible within the Möbius framework.

4.1 Execution policy review

Due to space limitations, we refer the reader to [18] for a more detailed description of the Möbius execution policy. To define actions, it is necessary to introduce some new terms and concepts.

An action may be *enabled* in a particular state or not, and when it is enabled, it performs work towards completion. An action completes and changes the state to reflect the condition of the completed work. Any *event* (state change) in which an action is enabled may be an interrupting event. A random variable describes the time between enabling and completion of an action, and the effort function describes the amount of work completed by an action over time.

Five variables are able to capture the state of an action: the start time, delay distribution, effort function, worker ef-

fort, and minimum task effort. At any event that is an enabling event, disabling event, or interrupting event, three independent choices are made: 1) whether to preserve or discard (set to zero) the worker effort, 2) whether to preserve or discard (set to zero) the minimum task effort, and 3) whether to preserve the delay and effort functions, or discard them and later choose new ones. This can be written as an acronym, e.g., PDP means preserve worker effort, discard minimum task effort, and preserve the delay and effort functions.

This execution policy allows us to implement all the various preemption policies preemptive resume (prs), preemptive repeat different (prd), and preemptive repeat identical (pri) described in [4], the concept of reactivation found in SANs [25], as well as other execution policies not generally considered by others (see [18] for examples of these). Möbius actions must be defined so that they can utilize this general execution policy. We provide the action definition in the following section and then give examples of how state-change mechanisms in particular formalisms can be expressed as actions.

4.2 Action components

A model has a set of actions A. An action is made up of three components: action functions, action state, and an initial action state. Action functions provide information to the execution policy prescribing how the action interacts with other model components. The action state provides the state information for an action. The initial action state provides an action with an initial state at time 0. We now describe each in more detail.

4.2.1 Action functions

Formally, an *action function* is the mapping

$$AF : A \rightarrow \mathit{Enabled} \times \mathit{Delay} \times \mathit{Effort} \times \mathit{Rank} \times \mathit{Weight} \times \mathit{Complete} \times \mathit{Interrupt} \times \mathit{Policy}\,,$$

where each of the terms in the co-domain is a function, whose type is given in Figure 4. We use the symbol Σ to denote the set of state variable values, and E to be the set of possible *events*. An event takes the form of the triple (Σ, a, Σ'): a state, the action that may complete in state Σ, and the state resulting from the completion of a in state Σ. Note that since we frequently talk about a single action function as opposed to the set of functions, we adopt an object-oriented style of notation, with which we write $a.\mathit{Enabled}$ to mean the $\mathit{Enabled}$ function of $AF(a)$.

We briefly describe each of the action functions.

$\mathit{Enabled} : \Sigma \rightarrow \mathit{bool}$ A Boolean function ($\{\mathit{true}, \mathit{false}\}$) that depends on the model state, and indicates when an action is enabled.

$\mathit{Delay} : \Sigma \rightarrow (\mathbb{R} \rightarrow [0,1])$ A distribution function of a random variable describing the uninterrupted time between enabling and completion of an action (or, in Petri net terms, firing of a transition).

$$\begin{array}{lll} \mathit{Enabled} & : & \Sigma \rightarrow \mathit{bool} \\ \mathit{Delay} & : & \Sigma \rightarrow (\mathbb{R} \rightarrow [0,1]) \\ \mathit{Effort} & : & \Sigma \rightarrow (\mathbb{R} \rightarrow [0,1]) \\ \mathit{Rank} & : & \Sigma \rightarrow \mathbb{Z} \\ \mathit{Weight} & : & \Sigma \rightarrow \mathbb{R}^{\geq} \\ \mathit{Complete} & : & \Sigma \rightarrow \Sigma \\ \mathit{Interrupt} & : & E \rightarrow \mathit{bool} \\ \mathit{Policy} & : & \Sigma \rightarrow \{\mathrm{DDD}, \ldots, \mathrm{PPP}\} \end{array}$$

Figure 4. Action functions

$\mathit{Effort} : \Sigma \rightarrow (\mathbb{R} \rightarrow [0,1])$ A function describing how work proceeds over time. This is discussed in greater detail in [18].

$\mathit{Rank} : \Sigma \rightarrow \mathbb{Z}$ A function used to arbitrate actions that are scheduled to complete at the same time. Higher-rank actions complete first.

$\mathit{Weight} : \Sigma \rightarrow \mathbb{R}^{\geq}$ A function used to select probabilistically among actions that are scheduled to complete at the same time and have the same rank. The probability that an action will complete is the weight of the action divided by the sum of the weights of competing actions. If the weight is zero, then the probability is assumed to be undefined and the well-specified algorithm [17] must be used.

$\mathit{Complete} : \Sigma \rightarrow \Sigma$ A function that provides the new state of the state variables when an action completes.

$\mathit{Interrupt} : \Sigma \rightarrow \mathit{bool}$ A Boolean function that yields true if an event occurs that is an interrupting event, e.g., a reactivation event in SANs. This is described in greater detail in [18].

$\mathit{Policy} : \Sigma \rightarrow \{DDD, \ldots, \mathrm{PPP}\}$ A function that describes which policy should be taken by the action in any enabling change or interrupting event. The codomain is the set $\{$DDD, DDP, DPD, DPP, PDD, PDP, PPD, PPP$\}$ and corresponds to one of the following choices: preserve or discard worker effort, preserve or discard minimum task effort, and preserve or discard the delay and effort functions. These policies are described in detail in [18].

4.2.2 Action state

The second component of an action is the action state. Formally, an *action state* is also a mapping. Let

$$AS : A \rightarrow \mathrm{Start} \times \mathrm{Delay} \times \mathrm{Effort} \times \mathrm{WE} \times \mathrm{MTE}\,,$$

where each element in the co-domain is a function given in Figure 5.

The action state, like state variable values, is not a model component. Rather, it is used in describing the execution and the underlying stochastic process of a model.

We describe each action state component below.

$$\begin{array}{lcl}
\text{Start} & : & \mathbb{R}^{\geq} \\
\text{Delay} & : & (\mathbb{R} \rightarrow [0,1]) \cup \{\emptyset\} \\
\text{Effort} & : & (\mathbb{R} \rightarrow [0,1]) \cup \{\emptyset\} \\
\text{WE} & : & [0,1] \\
\text{MTE} & : & [0,1]
\end{array}$$

Figure 5. Action state

$\text{Start} : \mathbb{R}^{\geq}$ The time of the last defining event.

$\text{Delay} : (\mathbb{R} \rightarrow [0,1]) \cup \{\emptyset\}$ A value from the distribution function describing the (random) uninterrupted time between enabling and completion of an action. This is the "sampled" value of $a.Delay$.

$\text{Effort} : (\mathbb{R} \rightarrow [0,1]) \cup \{\emptyset\}$ A value that describes the amount of work produced by the action over time. This is also the "sampled" value of $a.\mathit{Effort}$.

$\text{WE} : [0,1]$ A measure of the preserved work performed by the action up until the last defining event.

$\text{MTE} : [0,1]$ The minimum amount of work known to be required by the action at the instant of the last defining event.

These state variables are sufficient to capture the state of an action implementing the Möbius execution policy.

4.2.3 Initial action state

The initial action state is the same for all actions. In particular, the *initial action state* is $\text{Start} = 0$, $\text{Delay} = \text{Effort} = \emptyset$, and $\text{WE} = \text{MTE} = 0$.

4.3 Properties

Let $AP : A \rightarrow 2^{\Pi}$ be the *action properties* of a Möbius model. An example may be <exponential> $\in AP(a_i)$, which indicates that this action always has an exponential delay distribution; this is useful for Markov state-space generators. Other examples are <affects s_j> $\in AP(a_i)$, which indicates that when a_i completes, it may change the value of state variable s_j; and <pri> $\in AP(a_i)$, which means that a_i has the execution policy of preemptive repeat identical. This information can be used to make simulation or analytic solution more efficient.

4.4 Definition of actions

Finally, we can define the action components of a model as

$$Act = (A, AF, AP).$$

5 Atomic model definition

Let $MP : 2^{\Pi}$ be a set of *model properties* that are associated with a model.

We can now formally define an atomic model AM in the Möbius framework. Formally, an *atomic model* is a triple

$$AM = (SV, Act, MP)\,,$$

that is, a set of state variables, actions, and properties. Recall that $SV = (S, \mathit{type}, IS, P_S, SVP)$ is the set of state variables, including the names, types, and initial value distribution; $Act = (A, AF, AP)$ is the set of actions, including names and action functions; and MP is a set of symbols representing properties of the model. As defined in this paper, atomic models are the basic building blocks of models within the Möbius framework.

5.1 On properties

The vocabulary of properties is extensible. New symbols may be added as formalisms and solvers are added to the tool. If a solver encounters a symbol it does not understand, it may safely ignore it, because symbols are only used for increased efficiencies in solution. Composed model and reward variable formalisms may also use properties, and it is important that these formalisms understand which properties they may preserve. It is possible to maintain consistency safely by following the simple rule that any property that a formalism does not understand shall not be preserved by that formalism.

The use of properties implies a certain trust factor between formalisms and solvers. Solvers may rely extensively on properties, and it is up to the Möbius formalism designer to ensure that the properties of a model are true. Failure could result in inappropriate application of specialized solution methods.

Properties are the one area of Möbius in which components are not completely modular. A new formalism and solver may introduce new properties that are not understood by other composition, reward variable, and connection formalisms. In that case, each of the formalisms should be updated as to whether it preserves the new property. The rule that states that formalisms may not preserve properties that they do not understand keeps Möbius safe. However, until all model components are updated to respond appropriately to the property, users may not have access to the full modeling power of Möbius. We believe that the cost of updating each tool component to respond to new properties is minor compared to the effort required to implement a new formalism or solver.

6 Example

We illustrate the use of Möbius in capturing an SRN [8] atomic model. Let $\mathbf{A}$ be a SRN model. $\mathbf{A}$ is defined in the usual way:

$$\mathbf{A} = \{P, T, D^-, D^+, D^\circ, g, >, \mu_0, \lambda, w, M\}$$

where $P = \{p_i\}$ is a finite set of places, $T = \{t_i\}$ is a finite set of transitions, D^+, D^-, and D° are the marking-dependent multiplicities of the input, output, and inhibitor

arcs, g is the guard function, $>$ is the priority relation, μ_0 is the initial marking, λ is the rate function of the transitions, w is the weight function used for immediate transitions, and M is a set of measures.

State variables are constructed in the following way. Let $S = \{s_i\}$ be the state variables corresponding to places $\{p_i\}$. Let $\forall s_i \in S$, $type(s_i) = \{0, \dots, \infty\}$ by Rule 2. Let $IS = \{val_{IS}\} : val_{IS}(s_i) = \mu_0(p_i)$ with $P_S(val_{IS}) = 1$. We can define the equivalence $val_\sigma(s_i) = \#(p_i, \mu)$ where state σ corresponds to marking μ. This allows us to use val and $\#$ interchangeably.

Let $A = \{a_i\}$ be a set of Möbius actions corresponding to SRN transitions $T = \{t_i\}$. We define an auxiliary function $arcenabled : A \to bool$ to indicate whether the corresponding transition is arc-enabled. $arcenabled(a_i) = true$ in marking μ iff for the corresponding transition t_i

$$\forall p \in P, D^-_{p,t_i}(\mu) \leq \#(p, \mu) \wedge \left(D^\circ_{p,t_i}(\mu) > \#(p, \mu) \vee D^\circ_{p,t_i}(\mu) = 0\right)$$

The $Enabled$ function of action a_i is $true$ in state σ iff

$$arcenabled(t_i) \wedge g_{t_i}(\mu) \wedge \nexists t_j : t_j > t_i \wedge arcenabled(t_j) \wedge g_{t_j}(\mu)$$

where $>$ is the priority relation. $a_i.Delay(\sigma) = 1 - e^{\lambda(\mu)t}$ if $\lambda(\mu) < \infty$, otherwise it is the step function. Effort and rank are irrelevant, and $a_i.Weight(\sigma) = w_{t_i}(\mu)$. The $Complete$ function of a_i is calculated as follows. When transition t fires, the next marking μ' is

$$\forall p \in P, \#(p, \mu') = \#(p, \mu) - D^-_{p,t}(\mu) + D^+_{p,t}(\mu)\,,$$

and σ' is equivalent to μ'. Finally, $a_i.Interrupt = true$ and $a_i.Policy = \text{DDD}$.

7 Conclusion

The Möbius modeling tool [13, 14], while still a work in progress, has impressive features. Based on the framework, and, in particular, the atomic model described in this paper, we have implemented several atomic modeling formalisms, including SANs [20], PEPA [12], and the buckets and balls formalism [13] within the Möbius tool.

Other module implementations include a Replicate/Join composition formalism [33], a graph composition formalism of [28] (without symmetry detection), and a rate-impulse reward variable specification [34]. Solvers include a distributed simulator and Markov chain-based solvers. The key to the implementation is the use of the object-oriented features of C++ to make abstract base classes [20, 13]. We designed Möbius components as abstract classes, and formalisms derive formalism components from these base classes. In that way, other formalisms and solvers can manipulate the formalism components as framework components.

The choices and definitions of each Möbius model component make the Möbius framework possible. In this paper, we have described the major model components, including various model types, and given an overview of the whole framework. Then we focused in detail on atomic models, which are made up of actions, state variables, and properties. We described each of these in turn. With this work, we have created the basis of a tool that can incorporate multiple modeling formalisms and allow maximum interaction between modeling techniques.

References

[1] M. Ajmone Marsan, G. Balbo, and G. Conte. A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems. *ACM Trans. Comput. Syst.*, 2:93–122, 1984.

[2] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios. Open, closed, and mixed networks of queues with different classes of customers. *Journal of the Association for Computing Machinery*, 22(2):248–260, Apr. 1975.

[3] F. Bause, P. Buchholz, and P. Kemper. A toolbox for functional and quantitative analysis of DEDS. In R. Puigjaner, N. N. Savino, and B. Serra, editors, *Computer Performance Evaluation: Modelling Techniques and Tools: Proceedings of the 10th International Conference*, number 1469 in Lecture Notes in Computer Science, pages 356–359, Palma de Mallorca, Spain, Sept. 1998. Springer.

[4] A. Bobbio, A. Puliafito, and M. Telek. A modeling framework to implement preemption policies in non-Markovian SPNs. *IEEE Trans. Softw. Eng.*, 26(1):36–54, Jan. 2000.

[5] R. M. L. R. Carmo, L. R. de Carvalho, E. de Souza e Silva, M. C. Diniz, and R. R. R. Muntz. TANGRAM-II. In R. Marie, B. Plateau, M. Calzarossa, and G. Rubino, editors, *Computer Performance Evaluation; Modelling Techniques and Tools; Proceedings of the 9th International Conference*, number 1245 in Lecture Notes in Computer Science, pages 6–18, St. Malo, France, June 1997. Springer.

[6] G. Chiola, G. Bruno, and T. Demaria. Introducing a color formalism into generalized stochastic Petri nets. In *9th European Workshop on the Application and Theory of Petri Nets*, pages 202–215, Venice, Italy, June 1988.

[7] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. GreatSPN 1.7: GRaphical Editor and Analyzer for Timed and Stochastic Petri Nets. *Performance Evaluation*, 24(1–2):47–68, Nov. 1995.

[8] G. Ciardo, A. Blakemore, P. F. J. Chimento, J. K. Muppala, and K. S. Trivedi. Automatic generation and analysis of Markov reward models using stochastic reward nets. In C. Meyer and R. J. Plemmons, editors, *Linear Algebra, Markov Chains, and Queueing Models*, pages 141–191. Springer-Verlag, 1993.

[9] G. Ciardo and A. S. Miner. SMART: Simulation and Markovian analyzer for reliability and timing. In *Tool Descriptions from the 9th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (PERFORMANCE TOOLS '97) and the 7th International Workshop on Petri Nets and Performance Models (PNPM '97)*, pages 41–43, St. Malo, France, June 1997.

[10] G. Ciardo and K. S. Trivedi. A decomposition approach for stochastic reward net models. *Performance Evaluation*, 18:37–59, 1993.

[11] G. Ciardo and K. S. Trivedi. SPNP: The stochastic Petri net package (version 3.1). In *Proceedings of the 1st International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'93)*, pages 390–391, San Diego, California, USA, Jan. 1993.

[12] G. Clark. Incorporating stochastic process algebra into the Möbius framework. In *Proceedings of Process Algebra and Performance Modelling (PAPM 2001)*, Aachen, Germany, Sept. 2001.

[13] G. Clark, T. Courtney, D. Daly, D. D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster. The Möbius modeling tool. In *Proceedings of Petri Nets and Performance Models (PNPM 2001)*, Aachen, Germany, Sept. 2001.

[14] D. Daly, D. D. Deavours, J. M. Doyle, P. G. Webster, and W. H. Sanders. Möbius: An extensible tool for performance and dependability modeling. In B. R. Haverkort, H. C. Bohnenkamp, and C. U. Smith, editors, *Computer Performance Evaluation: Modelling Techniques and Tools: Proceedings of the 11th International Conference, (TOOLS 2000)*, number 1786 in Lecture Notes in Computer Science, pages 332–336, Schaumburg, IL, USA, Mar. 2000. Berlin: Springer.

[15] D. D. Deavours and W. H. Sanders. An efficient disk-based tool for solving large Markov models. *Performance Evaluation*, 33:67–84, 1998.

[16] D. D. Deavours and W. H. Sanders. "On-the-fly" solution techniques for stochastic Petri nets and extensions. *IEEE Trans. Softw. Eng.*, 24(10):889–902, Oct. 1998.

[17] D. D. Deavours and W. H. Sanders. An efficient well-specified check. In P. Bucholz and M. Silva, editors, *Proceedings of the 8th International Workshop on Petri Nets and Performance Models (PNPM '99)*, pages 124–133, Zaragoza, Spain, Sept. 1999.

[18] D. D. Deavours and W. H. Sanders. The Möbius execution policy. In *Proceedings of Petri Nets and Performance Models (PNPM 2001)*, Aachen, Germany, Sept. 2001.

[19] S. Donatelli. Superposed generalized stochastic Petri nets: Definition and efficient solution. In R. Valette, editor, *Application and Theory of Petri Nets 1994, Lecture Notes in Computer Science 815 (Proc. 15th International Conference on Application and Theory of Petri Nets, Zaragoza, Spain)*, pages 258–277. Springer-Verlag, June 1994.

[20] J. M. Doyle. Abstract model specification using the Möbius modeling tool. Master's thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, Illinois, USA, 1999.

[21] R. Fricks, C. Hirel, S. Wells, and K. Trivedi. The development of an integrated modeling environment. In *Proceedings of the World Congress on Systems Simulation (WCSS '97)*, pages 471–476, Singapore, Sept. 1997.

[22] B. R. Haverkort. Performability evaluation of fault-tolerant computer systems using DyQN-Tool^{+}. *International Journal of Reliability, Quality, and Safety Engineering*, 2(4):383–404, 1995.

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

[24] P. Kemper. Numerical analysis of superposed GSPNs. In *Sixth International Workshop on Petri Nets and Performance Models (PNPM '95)*, pages 52–61, Durham, North Carolina, USA, Oct. 1995.

[25] J. F. Meyer, A. Movaghar, and W. H. Sanders. Stochastic activity networks: Structure, behavior, and application. In *Proc. International Workshop on Timed Petri Nets*, pages 106–115, Torino, Italy, July 1985.

[26] M. K. Molloy. Performance analysis using stochastic Petri nets. *IEEE Transactions on Computers*, 31:913–917, Sept. 1982.

[27] W. D. Obal II and W. H. Sanders. State-space support for path-based reward variables. In *Proceedings of the 3rd Annual IEEE International Computer Performance and Dependability Symposium (IPDS '98)*, pages 228–237, Durham, North Carolina, USA, Sept. 1998.

[28] W. D. Obal II and W. H. Sanders. Measure-adaptive state-space construction methods. *Performance Evaluation*, 44:237–258, Apr. 2001.

[29] B. Plateau and K. Atif. A methodology for solving Markov models of parallel systems. *IEEE Journal on Software Engineering*, 17(10):1093–1108, 1991.

[30] R. J. Pooley. The integrated modelling support environment: A new generation of performance modelling tools. In G. Balbo and G. Serazzi, editors, *Computer Performance Evaluation; Modelling Techniques and Tools; Proceedings of the 5th International Conference*, pages 1–15, Torino, Italy, Feb. 1991. Amsterdam: Elsevier.

[31] R. A. Sahner, K. S. Trivedi, and A. Puliafito. *Performance and Reliability Analysis of Computer Systems, An Example-Based Approach Using the SHARPE Software Package*. Kluwer, Boston, 1996.

[32] W. H. Sanders. Integrated frameworks for multi-level and multi-formalism modeling. In *Proceedings of the 8th International Workshop on Petri Nets and Performance Models*, pages 2–9, Zaragoza, Spain, Sept. 1999.

[33] W. H. Sanders and J. F. Meyer. 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.

[34] W. H. Sanders and J. F. Meyer. A unified approach for specifying measures of performance, dependability, and performability. In A. Avizienis, J. Kopetz, and J. Laprie, editors, *Dependable Computing for Critical Applications*, volume 4 of *Dependable Computing and Fault-Tolerant Systems*, pages 215–237. Springer-Verlag, 1991.

[35] W. H. Sanders, W. D. Obal II, M. A. Qureshi, and F. K. Widjanarko. The *UltraSAN* modeling environment. *Performance Evaluation*, 24(1):89–115, Oct. 1995.

[36] C. H. Sauer and E. A. MacNair. *Simulation of Computer Communication Systems*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1983.

[37] A. P. A. van Moorsel and Y. Huang. Reusable software components for performability tools, and their utilization for web-based configuration tools. In R. Puigjaner, N. N. Savino, and B. Serra, editors, *Computer Performance Evaluation; Modelling Techniques and Tools; Proceedings of the 10th International Conference*, number 1469 in Lecture Notes in Computer Science, pages 37–50, Palma de Mallorca, Spain, Sept. 1998. Springer.