The Möbius Framework and Its Implementation

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

Abstract— **The Möbius framework is an environment for supporting multiple modeling formalisms and solution techniques. Models expressed in formalisms that are compatible with the framework are translated into equivalent models using Möbius framework components. This translation preserves the structure of the models, allowing efficient solutions. The framework is implemented in the tool by a well-defined abstract functional interface. Models and solution techniques interact with one another through the use of the standard interface, allowing them to interact with Möbius framework components, not formalism components. This permits novel combinations of modeling techniques, and will be a catalyst for new research in modeling techniques. This paper describes our approach, focusing on the "atomic model." We describe the formal description of the Möbius components as well as their implementations in our software tool.**

Keywords— **Stochastic models, modeling formalisms, modeling frameworks, modeling tools, Markov models, stochastic Petri nets, PEPA, execution policy.**

I. Introduction

Performance and dependability modeling is an integral 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; queuing 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, formalisms (or formal languages for expressing models) were also developed. Each formalism has its own merits. Some formalisms afford very efficient solution methods; for example, BCMP [1] queuing networks admit product-form solutions, while superposed generalized stochastic Petri nets (SGSPNs) [2] afford Kronecker-based solution methods, and colored GSPNs (CGSPNs) [3] yield state-space reductions. Other formalisms, such as SPNs [4] and SPAs [5], provide a simple elegance in their modeling primitives, while a number of extensions, such as stochastic activity networks (SANs) [6], 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. [7] lists a number of such tools, such as DyQN-Tool+ [8], which uses dynamic queuing networks as its high-level formalism; GreatSPN [9], which is based on GSPNs [10]; *UltraSAN* [11], which is based on SANs [6]; SPNP [12], which is based on stochastic reward networks [13]; and TANGRAM-II [14], which is an object- and message-based formalism for evaluating computer and communication systems. While all of these tools are useful within the domains 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. Thus, it is difficult to model systems that cross different domains and would benefit from multiple modeling techniques.

A. Related Work

We will now briefly review the tools that are most closely related to Möbius; a more complete discussion of related work can be found in [7].

One approach has been what we call the "integrated software environment" approach. This approach seeks to unify several different modeling tools of the kind described above into a single software environment. Examples are IMSE (Integrated Modeling Support Environment) [15], IDEAS (Integrated Design Environment for Assessment of Computer Systems and Communication Networks) [16], and Freud [17]. 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.

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 [18]. In the SHARPE modeling framework, models can be expressed as combinatorial reliability models, directed acyclic

D. D. Deavours is with the Information and Telecommunication Technology Center, University of Kansas at Lawrence. Email: deavours@ittc.ku.edu

G. Clark is a Software Engineer with Citrix Systems, Inc. Email: grahamc@citrix.com

T. Courtney, D. Daly, S. Derisavi, and W. H. Sanders are with the Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign. Email: {tod, ddaly, derisavi, whs}@crhc.uiuc.edu

J. M. Doyle is a Software Engineer with Honeywell in Fort Washington, PA. Email: jay.doyle@honeywell.com

P. G. Webster is a Design Engineer at ARM, Inc. in Austin, TX. Email: pwebster@arm.com

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 NSF or Motorola.

task precedence graphs, Markov and semi-Markov models, product-form queuing networks, or GSPNs. 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 [19]. SMART supports the analysis of models expressed as SPNs and queuing 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 [20] 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.

B. The Möbius Approach

We take an integrated multi-formalism multi-solution approach with Möbius [7]. Our goal is to build a tool in which each model formalism or solver is, to the extent possible, modular, in order to maximize potential interaction. A modular modeling tool 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 [21] 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 labelled transition system, such as PEPA [22].

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 into prototypes (for example, as described in [23–27]), and are not generally available to other users. In particular, we would have liked to develop a new composition formalism called "graph composition" [27], but found that it would have been difficult to include it in *UltraSAN* because of the inherently closed nature of the software design. Möbius has been designed to include these prototyped modeling techniques, as well as to be extensible to include future techniques.

The ability to add new components simply would benefit researchers and users alike. Researchers would be able to add a new component to the tool and expect it to be able to interact immediately with other components. They would also be able to compare competing techniques directly. Additionally, researchers would have access to the work of others, and be able to extend and compare techniques. Users would benefit by having access to the most recent developments in conjunction with previously existing techniques. They would also benefit from having a modular, "toolbox" approach that would allow them 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, and Möbius should preserve such features.

For all of the above reasons, we argue that an open, multi-formalism, multi-solution modeling framework would represent a significant step forward in advancing the state-of-the-art in performance/dependability modeling techniques. Such a framework would maximize potential interaction of techniques, while maintaining the independence of those techniques. This would require the creation of a sufficiently general and abstract representation of a model that also retained the ability to be solved efficiently. Möbius is our attempt at a multi-formalism, multi-solution modeling framework for discrete event stochastic systems. Möbius should support existing formalisms as well as formalisms to be developed in the future. Therefore, we make no claim or proof that ours is the best or only solution, but instead offer several examples supporting the claim that it is a good solution.

In this paper, we present an overview of the Möbius framework and tool, and describe the "atomic model" and its implementation issues in greater detail. The paper is intended to give a view of the issues involved in developing the tool and in implementing new formalisms.

II. Möbius Overview

The Möbius framework is an environment for supporting multiple modeling formalisms. In order for a formalism to be compatible with the framework, the formalism must be able to translate any formalism model into an equivalent model that uses Möbius framework components. Since models are constructed in the specific formalisms, the expressive advantages of the particular formalisms are preserved. Because all models are transformed into framework components, all models and solution techniques in the framework are able to interact with each other. The framework is also extensible, allowing new formalisms and solvers to be added with little impact on existing ones, since new formalisms and solvers communicate using framework components.

In order to accomplish the desired goal of extensibility, framework components must be general enough to express a variety of different formalism components. However, there is a subtle but important point concerning the Möbius framework: it is not meant to be 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 experienced researchers to accomplish various purposes.

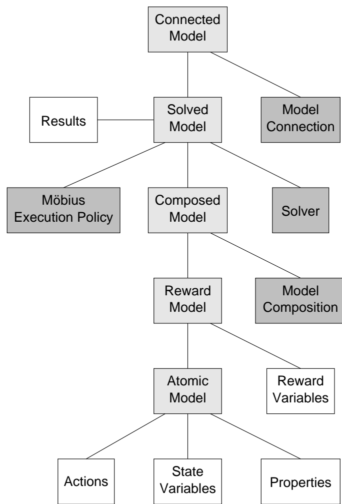


Fig. 1. Möbius framework components.

A. Framework components

In order to define the framework, we must identify and abstract the common concepts found in most formalisms. We also must generalize the process of building and categorizing models. We divide the model construction process into several steps. Each step in the process generates a new type of model. The illustration shown in Figure 1 highlights the various model types and other components within the Möbius framework.

The first step in the model construction process is to generate a model using some formalism. This most basic model in the framework 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, or queues in queuing networks) hold state information about a model, while actions (such as transitions in SPNs or servers in queuing networks) are the mechanism for changing model state. Properties provide information about a model that may be needed to allow use of a specialized solver, or to make the solution process more efficient for some solvers.

After an atomic model is created, frequently the next step is to specify some measures of interest on the model using some reward specification formalism, e.g., [28]. The Möbius framework captures this pattern by having a separate model type, called *reward 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 reward 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 include the Replicate/Join composition formalism [21] and the graph composition formalism of [27], in which symmetries may be detected and state lumping may be performed. Another notable composed model technique is synchronization on actions, which is found, for example, in stochastic process algebras (SPAs) such as PEPA [5], as well as in stochastic automata networks (and also SANs, e.g., [29]) and superposed GSPNs (e.g., [2, 30]). Although a composed model is a single model with its own state space, it is not a “flat” model. It is hierarchically built from submodels, which largely preserve their formalism-specific characteristics so the composed model does not destroy the structural properties of the submodels. 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. Composition of different formalisms (specifically SAN and PEPA) is demonstrated by example models included in the Möbius distribution. Additionally, we note that reward variables can also be added to a composed model.

The next step is typically to apply some solver to compute a solution and generate a *solved model*. We call any mechanism that calculates the solution to reward variables a *solver*. The calculation method could be exact, approximate, or statistical, and it may take advantage of model properties that are due to the atomic model formalisms, model composition, and reward specification. 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 reward models that together form a connected model.

A *connected model* is an ordered set of reward models and their corresponding solution methods 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 [31]. In those cases, the model of interest is a set of reward models with dependencies expressed through results, where the overall model may be solved through a system of nonlinear equations (if a solution exists).

B. Tool description

The Möbius tool is our implementation of the Möbius framework. It ensures that all formalisms translate model components into framework components through the use of the abstract functional interface (AFI) [32]. The AFI pro-

vides the common interface between model formalisms and solvers that allows formalism-to-formalism and formalism-to-solver interactions. It uses abstract classes to implement Möbius framework components. The AFI is built from three main base classes: one for state variables, one for actions, and one that defines overall atomic model behavior. Each of these classes defines an interface used by the Möbius tool when building composed models, specifying reward variables, and solving models. There are subtle differences between the current AFI and the framework, predominantly due to delays in the implementation of newer framework concepts (such as properties), and additions to the AFI to increase efficiency or ease of use of the tool.

The various components of a model formalism must be presented as classes derived from the Möbius AFI classes in order to be implemented in the Möbius tool. Other model formalisms and model solvers in the tool are then able to interact with the new formalism by accessing its components through the Möbius abstract class interfaces.

The main user interface for the Möbius tool presents a series of editors that are classified according to model type. Each formalism or solver supported by Möbius has a corresponding editor in the main interface. These editors are used to construct and specify the model, possibly performing some formalism-specific analysis and property discovery, and to define the parameters for the solution techniques. The tool dynamically loads each formalism-specific editor from a java archive (`jar` file) at startup. This design allows new formalisms and their editors to be incorporated into the tool without modification or recompilation of the existing code, thus 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 packaged to form a library archive. These libraries are linked together along with the tool's base libraries to form the executable for the solver. The executable is run to generate the results. The base libraries implement the 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 2.

We believe that the majority of modeling techniques can be supported within the Möbius framework and tool. 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. That approach also makes it possible for the framework to be extensible, in that new atomic modeling formalisms, reward formalisms, compositional formalisms, solvers, and connection formalisms may be added independently. All of these features will be discussed in more detail in the remaining sections. Special focus will be given to the atomic model, since it represents a generalization of multiple modeling formalisms and is one of the main contributions of Möbius. The key elements of atomic models are state variables and actions; they are the subjects of the next two sections.

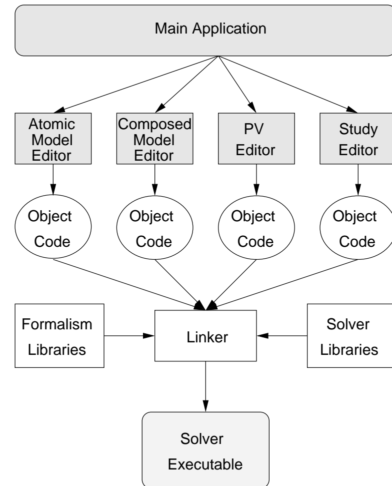


Fig. 2. Möbius tool architecture

III. State variables

A state variable typically represents some portion of the state of a model, and is a basic component of a model. It can represent something 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 set of values that a place can take on is the set of natural numbers. Colored GSPNs (CGSPNs) [3] 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. Queuing networks with different customer classes can have more complicated notions of state, such as those found in extended queuing networks [33], in which each job (customer) may have an associated job variable, which is typically implemented as an array of real numbers.

To capture and express all state variable types in existing formalisms in Möbius, we must create a generalized state variable that can be used to create specific state variables. By using a generalized state variable, we enjoy all the benefits of a framework we discussed earlier. Specifically, solvers or higher-level model types can interact with Möbius state variables (in the framework or the tool), instead of with the variety of different formalism state variables. Finally, any efficiencies that may be gained through any structural knowledge can be preserved through the use of properties.

A. Components of State Variables

In the Möbius framework, a state variable is made up of three primary components: a value, a type, and an initial value distribution, which includes a set of possible initial states. The value of a state variable 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

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 $\cup_{i=1}^{n} t_i \in T$.

Fig. 3. Construction of the type set T.

time zero. A distribution of values allows us to consider models whose initial states may vary. For example, we could consider a system that is in steady state, and measure the expected time until some important event occurs.

A.1 Types

We begin by describing state variable types in the Möbius framework. 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 $type : S \to T$ 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, S is the set of state variables, 2^t is the power set of t, and ν is a `nil` element. The meaning of $([a, b)] \in T$ is $[a, b], (a, b], [a, b), (a, b) \in T$. Examples of types are given in Section III-C.

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

A.2 Values

Naturally, state variables take on values, and the values of state variables change over time in the Möbius framework. 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 $val : S \to V$ such that $val(s) \in type(s)$, where $V \in T$. The value function is not a formal element of the model in the Möbius framework, but is used in describing the execution of a model.

A.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, the Möbius framework includes a set of possible initial values and a probability function describing the probability of being in each initial value. Let $IS = \{val\}$ be a countable set of initial values for the state variables. Let $P_S : IS \to [0, 1]$ be the initial value probability distribution so that $P_S(v) = \Pr[\text{Initial value of state variables is } v]$. This allows the initial state of state variables to be probabilistically distributed among different values.

A.4 Properties

Properties are a set of symbols that specify that a certain condition or conditions about a Möbius model component are true in the Möbius framework. 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.

A property is nothing more than a symbol that has a proprietary meaning. We write each property in the form <property>. Let Π be the set of all properties. The *state variable properties* are a function $SVP : S \to 2^{\Pi}$. 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.

A.5 Definition

Now we can formally define the *state variable components* of a model in the Möbius framework to be the components

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

B. State Variable Implementation

The implementation of the state variables requires the development of a C++ class to represent the core functionality of a state variable. All specific state variable types will inherit this base class. Therefore, the base class for Möbius state variables, `BaseSV`, specifies the minimal interface that any state variable must provide in the Möbius tool, and it implements a subset of the functionality that state variables have in the framework. At the current time properties are not yet supported for state variables, and each state variable can have only one initial value. 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 [32]). 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`, that are implicitly in the framework. The methods `AffectingActs` and `EnablingActs` link state

TABLE I

Selection of Möbius AFI functions for state variables

Function
`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.

variables with actions in the tool. They are useful because in practice, model events often have only a "local" impact on model state. This local impact could be reflected using action properties, but because the local impact may have a significant effect on solution efficiency, those two methods are included in the AFI. The model formalism implementor may be able to implement the methods intelligently such that depending on the model, only a subset of model actions will be linked to any state variable, allowing more efficient solutions.

C. Examples

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

A GSPN [10] 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. A computer cannot handle infinite values, so the implementation of a GSPN could use an `unsigned short` or `int` type to represent a GSPN place.

The value of a CGSPN [3] place can be expressed using a set of pairs: a color and the number 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. Similarly, C can be represented by an enumerated data type `enum colors {color1, color2, ..., color`n`}` in an implementation. The number of tokens of a color in s is $\mathbb{N} \in T$, as shown above, and could be implemented as an `unsigned short`. 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. CGSPN state can be implemented as a `struct` containing one element of type `color` (defined above) and one of type `unsigned short`.

SPAs are significantly different from SPNs because state variables are not explicitly expressed in SPAs. For example, consider a PEPA [5] model. Its 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 $1, \dots, K$. Note that $\{1, \dots, K\} \in T$ by rule 2. The sequential stages can easily be implemented as an enumerated type `enum sequential {stage1, stage2, ..., stage`K`}`, or as an `unsigned short`. A more clever approach can be found in [22]. 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 [1]. Let C be the set of integers that enumerate the set of customer classes. We know that $t_1 = \{0\} \cup C \in T$ can be implemented as one enumerated type `enum classes {null, class1, ..., class`n`}`. 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. The tool currently supports only finite arrays, so t_2 must be implemented as an array of type `classes`, as defined above.

Finally, we illustrate the extended queuing network (EQN) [33] job variable. A job variable is an array of k real numbers, and there is one job variable associated with each job. Using rule 7, we know that $t_1 = \mathbb{R} \times \cdots \times \mathbb{R} \in T$ (a k-tuple), which can be implemented as an array of size k of type `float`. Again, using rule 2, we know $C \in T$, where C is the set of integers that enumerate the customer classes (colors), and can be implemented as the enumerated type `colors` defined for the CGSPN. Using rules 5 and 7, we know $t_2 = S \cup \{\nu\} \times C \in T$, which can be implemented as a `struct` containing one element of type `color` and a pointer to state variable type t_1. 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, which can be implemented as a finite array. 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 and implement it. We chose only one for illustration.

IV. Actions

An action is the basic model unit that changes the value of state variables in the Möbius framework, and is therefore the basic model unit that changes model state. An action corresponds to a transition in SPNs [4], GSPNs [10], and other extensions, to an action of an SPA (e.g., [5]), to an activity of a SAN [6], or to a server of a queuing network (e.g., [1]), for example.

Actions are similar to state variables in the framework, 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 queuing 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 [34], 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 below the execution policy of the Möbius framework.

Finally, like state variables, the action provides a common interface by which other model components (possibly of different formalisms) and solvers may interact in the Möbius framework. This allows for composition by synchronization, as is found in SPAs, stochastic automata networks (e.g., [29]), and superposed GSPNs (e.g., [2,30]).

We begin with the description of the action components, including a discussion of the execution policy, before giving a formal definition of an action. We then show how actions are implemented in the tool.

A. Action Components

A model has a set of actions A in the Möbius framework. An action is made up of three primary components: action functions, action state, and action properties. 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, which is necessary for non-Markovian models. The action properties may be used by solvers to aid in efficient solution techniques. We now describe each in more detail.

A.1 Action Functions

Formally, in the Möbius framework an *action function* is the mapping

$$\begin{aligned} AF : A \rightarrow & \; \mathit{Enabled} \times \mathit{Delay} \times \mathit{Effort} \times \mathit{Rank} \\ & \times \mathit{Weight} \times \mathit{Complete} \times \mathit{Interrupt} \times \mathit{Policy}\,, \end{aligned}$$

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, E to be the set of possible *events*, and $\mathbb{R}^{\geq}$ to be the set of non-negative real numbers. An event takes the form of the 4-tuple $(\sigma, \tau, a, \sigma')$: a state, the sojourn time, the action that may complete in σ, and the state resulting from the completion of a in σ. Note that since we frequently talk about a single function (as opposed to all of the functions) of the action function, 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 in the Möbius framework.

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

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

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

Fig. 4. Action functions

$\mathit{Effort} : \Sigma \rightarrow (\mathbb{R}^{\geq} \rightarrow [0,1])$ A function describing how work proceeds over time. This is discussed in greater detail in [35], and briefly in Section IV-A.2.

$\mathit{Rank} : \Sigma \rightarrow \mathbb{Z}$ 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 of the same action partition group (see Section IV-A.4).

$\mathit{Complete} : \Sigma \rightarrow \Sigma$ A function that provides the new value 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 [35].

$\mathit{Policy} : \Sigma \rightarrow \{\mathrm{DDD}, \ldots, \mathrm{PPP}\}$ A function that describes which policy should be taken by the action in any enabling change or interrupting event. The co-domain is the set {DDD, DDP, DPD, DPP, PDD, PDP, PPD, PPP} and corresponds to one of the following choices: preserve or discard worker effort (WE), preserve or discard minimum task effort (MTE), and preserve or discard the delay and effort functions. These policies are described in detail in [35].

The functions *Effort*, *Interrupt*, and *Policy* are needed to implement the Möbius execution policy. When an enabled action is interrupted, three independent decisions need to be made, according to the Möbius execution policy: 1) whether to preserve or discard (set to zero) the action state WE (described in the following section), 2) whether to preserve or discard (set to zero) the action state MTE, and 3) whether to preserve the delay and effort functions, or discard them and later choose new ones. The set of three decisions can be written as an acronym, e.g., PDP means preserve WE, discard MTE, and preserve the delay and effort functions. The *Policy* function determines the answer to the three questions. Because to space limitations, we refer the reader to [35] for a more detailed description of the Möbius framework execution policy.

The execution policy allows us to implement in the Möbius framework all the various preemption policies: the preemptive resume (prs), preemptive repeat different (prd),

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

Fig. 5. Action state

and preemptive repeat identical (pri) described in [34]; the concept of reactivation found in SANs [6]; and other execution policies not generally considered by others (see [35] for examples). Actions must be defined so that they can utilize this general execution policy. We provide the action definition in the following sections.

A.2 Action State

The second component of an action in the Möbius framework is the action state. The state of the state variables, together with the action state, make up the state of a model. Formally, an *action state* is also a mapping. Let

$$AS : A \to \text{Start} \times \text{Delay} \times \text{Effort} \times \text{WE} \times \text{MTE}\,,$$

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 in the framework. Rather, it is used in describing the execution and the underlying stochastic process of a model (see [36]).

We describe each action state component below.

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

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

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

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

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

Here, we use the term *defining event*, for an action, to be any event in which the action becomes enabled, becomes disabled, or is interrupted. The above variables are sufficient to capture the state of an action implementing the Möbius framework execution policy.

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

A.3 Action Properties

The third component of an action is its action properties. Let $AP : A \to 2^{\Pi}$ be the *action properties* of a model in the framework. 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.

A.4 Action Partition Groups

The set of actions is partitioned into subsets of A called *action partition groups*. The purpose of action partition groups is to identify the set of actions that compete by weight if they have the same completion time and rank. The action weights are used to compute the relative probability of an action completing before other actions in the same action partition group. This allows some forms of non-determinism to be expressed, such as the non-determinism expressed by extended conflict sets in GSPNs and multiple enabled instantaneous activities in SANs. Formally, let the action partition groups, $AG : 2^{2^A}$, be a partition of the actions so that $\cup_{G \in AG} G = A$ and $\cap_{G \in AG} G = \emptyset$.

Of all the enabled actions, the action that will complete next is the one with the earliest completion time. The completion time is generally a random variable, which is the current execution time plus some "clock" value (described formally in [36]). If two or more actions have the same completion time, the action with the highest rank completes first.

If two or more actions with the same completion time share the same highest rank, then the process becomes more involved. Weights are used to select probabilistically among the remaining actions, but weights are only used to describe the relative probabilities of actions in the same action partition group. Among actions of different action partition groups, the relative probability of selecting actions is left unspecified. An efficient algorithm exists to determine whether the unspecified order is a problem [25]. Action partition groups are useful for capturing the behavior of SAN immediate activities and extended conflict groups, in which some level of unspecified ordering in action completions is possible.

A.5 Definition of Actions

Finally, we can define the *action components* of a model in the Möbius framework as

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

B. Action Implementation

The base class for Möbius actions in the Möbius tool, `BaseAction`, determines the characteristics of model actions, and implements a subset of the Möbius framework action functionalities. Table II lists a selection of the AFI functions associated with Möbius actions. The method `Fire` is the implementation of the *Complete* action function from the framework. It must be implemented for every

TABLE II
SELECTION OF MÖBIUS AFI FUNCTIONS FOR ACTIONS

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

action in a given model. The methods `Enabled`, `Rank`, and `Weight` directly correspond to the action functions of the same names.

The `SampleDistribution` method is used to determine the time after which an enabled action may complete. It is related to the *Delay* action function in that it takes one sample from the distribution returned by *Delay*. Other methods exist in the AFI to completely determine the value of the delay action function for any model state.

`ReactivationPredicate` and `ReactivationFunction` are methods that differ from the action functions of the framework. They currently implement a more limited execution policy than is present in the Möbius framework. The use of those two methods allows an action to either maintain its time to completion, or generate a new time to completion based on the current state when an action is interrupted.

Actions are linked back to state variables with the two methods `AffectedSVs` and `EnablingSVs`. These two methods do not correspond to any action function, but increase efficiency in solution by identifying which state variables play a role in enabling this action, and which ones are modified when this action fires. The methods could be formally represented using action properties, but because of their importance to efficient model solution, they are included in the base classes.

V. ATOMIC MODEL AND FORMALISMS

Based on the definitions of state variables and actions given in the previous sections, we will now formally define atomic models. We will follow the definition with a discussion of how model properties can be utilized to solve models efficiently. Finally, we will show how conceptually different atomic model formalisms are implemented in the Möbius tool based on the AFI. Particularly, we demonstrate how to realize Petri net model formalisms of different complexities and also PEPA$_k$, an SPA formalism.

A. Atomic Model Definition

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

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, state variable components, action components, and properties. Recall that the set of state variables is $SV = (S, type, IS, P_S, SVP)$, including the names, types, initial value distribution, and properties; $Act = (A, AF, AG, AP)$ is the set of actions, including names, action function, action partition group, and properties; 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.

B. On Properties

The vocabulary of properties in the framework is extensible. New symbols may be added as new formalisms and solvers are used with the framework. If a solver encounters a symbol it does not understand, it may safely ignore it, because symbols are used only for increased efficiencies in solution. Composed model and reward variable formalisms may also use properties, and it is important that a formalism that is operating on a model defined in another formalism understands which properties can be preserved. 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.

Properties are the one area of the Möbius framework 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. If that happens, each of the formalisms should be updated as to whether it preserves the new property. We believe that the cost of this updating is minor compared to the effort required to implement a new formalism or solver.

C. Atomic Model Implementation

The Möbius framework atomic model is implemented by the Möbius tool base class `BaseModel`, which is part of the AFI. Table III lists some of the methods that must be provided for any model of a formalism implemented in the Möbius tool. `BaseModel` provides functions that are used in both model composition and model solution. 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, such as `CurrentState`.

The Möbius tool uses an explicit component-based system for describing models. This gives a good match to component-based modeling formalisms, including SPNs,

TABLE III
SELECTION OF MÖBIUS AFI FUNCTIONS FOR MODELS

`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 model states are equivalent.
`List⟨BaseAction⟩* ListActions()`: Returns a list of all model actions.
`List⟨BaseSV⟩* ListSVs()`: Returns a list of all model state variables.

SANs, Markov processes, and queuing networks. In the next section, we show how arbitrary Petri net-based models are mapped into the AFI. It should be clear from this mapping that these formalisms can also easily be mapped onto the Möbius framework. The AFI has also proven to be general enough to allow the implementation of a stochastic process algebra formalism, namely PEPA$_k$, and we summarize how this is done in a later section.

D. Realizing Petri Net-Style Formalisms

We begin with an example using a series of more complicated Petri net formalisms. First we show how an SPN can be mapped to the framework and implemented, and then we address the extra features in GSPNs and SANs.

We consider an SPN classically, as a bipartite directed graph consisting of *places*, *transitions* with associated exponentially distributed completion times, and connecting arcs. In order to map an SPN to the Möbius AFI, the most intuitive method associates a state variable with every 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`. That would provide 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. The Möbius tool requires a fixed-size representation for each state, and an appropriate data type for an SPN place may be an `unsigned short`. Using that representation, it is easy to implement the methods `SetState` and `CurrentState`. In practice, class `Place` provides a method called `Mark`, which returns a C++ *reference*, that can be used to copy or change the value of the state variable. The AFI for state variables also provides methods `AppendAffectingAction` and `AppendEnablingAction`, which are used to add affecting and enabling actions during initialization of a model.

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.

GSPNs extend SPNs with the addition of immediate transitions, with which priorities and weights are associated. Immediate transitions map to actions with a constant zero delay distribution, while the priorities and weights map to the rank and weight action functions.

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. This can be captured by considering each (action, case) pair as an action itself, and placing all such pairs containing the same action in an action partition group.
- 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 any state, in order for connected activities to be enabled in that 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*. Gate predicates can be smoothly incorporated using the `Enabled` method, and gate functions by using the `Fire` method.
- 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 IV.

Consider Figure 6. An arriving customer is probabilistically designated a class. Instantaneous activity $\texttt{Choose}_i$ is enabled if place $\texttt{Class}_i$ is marked and the predicate of gate $\texttt{C}_i\texttt{serv}$ is true, where the intention is to give priority to Class 1 customers. The predicate for gate $\texttt{C}_2\texttt{serv}$ 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.

We entered the model presented in Figure 6 into the Möbius tool to demonstrate its implentation; then, as described in Section II-B, C++ classes that inherit from the AFI base classes, and that represent the model and its components, were automatically generated by the tool. Figure 7 shows the header file, including a class definition of `SANModel`, which inherits from `BaseModel`. (All code presented has been cut down manually for the sake of brevity.) Notice that a class is created for activity **Depart**, with declarations for each method that must be implemented. Within its scope, each activity has pointers to the state variables that its methods, such as `Enabled` and `Fire`, will need to access. Figure 8 shows parts of the generated C++ model source code. The constructor for the model base

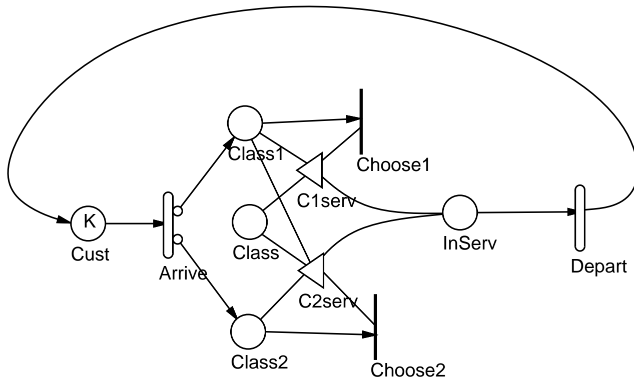


Fig. 6. A SAN representing an $M/M/1/K/K$ queue variation

```
#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. 7. C++ header file for model in Figure 6

```
#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. 8. C++ source code for model in Figure 6

class declares objects to represent state variables, and then uses AFI methods to initialize the enabled and affected action data structures for the state variables. Figure 8 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 the tool simply includes the gate predicate supplied by the modeler. For the `Fire` method, the tool again uses a combination of a state change determined by the structure of the net, and a state change specified by the user in the input gate function.

E. Realizing PEPA$_k$

The Möbius tool also supports an alternative style of modeling by providing PEPA (Performance Evaluation Process Algebra [5]), an SPA, as another atomic model formalism. Formally, PEPA models are specified in terms of a simple algebra. PEPA extends classical process algebra with the capacity to assign rates to activities, leading to the definition of a stochastic process. PEPA has been applied to the modeling of, for example, the performance of distributed computer systems, and components of a flexible manufacturing system [37]. In contrast to the graph-based approach of Petri nets, building PEPA models is analogous to writing programs. More details on incorporating PEPA into the Möbius tool can be found in [22].

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 richer mapping to the Möbius AFI, we enhanced PEPA with some convenient modeling features to produce PEPA$_k$, detailed in [22]. A simple example model illustrating these extensions is given in Figure 9. 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 $(\texttt{in}, \mu)$. Process parameters may also be communicated between activities, and the values may be 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 in the tool. The

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

Fig. 9. A queue with customer registration modeled as a PEPA_k sequential component

visible state variables of $Queue[m, n]$ are $\{m, n\}$. The state of the whole composed model is given by a vector of the states of each sequential component; the state of a sequential component is given by the values of its parameters and also by its 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 example, for the sequential component model in Figure 9, 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. Composed model state is a vector of sequential component state, where state vectors with particular order differences are identified according to the algorithm detailed in [38].

The 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. (If the model features two different copies of an activity of type `a`, they are distinguished as actions.) For the model above, the actions would be given by the set $\{\texttt{in}, \texttt{out}, \texttt{ref}\}$. If our model of interest was a cooperation between several components, then actions would be created for each possible synchronization of PEPA activities. See [22] for more details.

VI. Higher-level Model Types and Model Solution

After the modeler has made a number of atomic models as the building blocks of a large, complex model, she/he should be able to combine these submodels in order to construct the whole model. The next step is to define a set of measures on the model of interest. Finally, several solution methods should exist to compute the value of the measures and how they are affected by the changes in the model parameters. In this section, we describe these essential features as realized in various parts of the framework and the tool.

A. Composed Models

The Möbius framework 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 and then placing them together in a well-defined manner to construct a model of a system. One composition method is the *action-sharing* approach, in which submodels are composed through superposition of a subset of their actions. Another method is the *state-sharing* approach, which links submodels together by identifying sets of state variables. For example, it is possible to compose two Petri net models by causing them to hold 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. Note that a composition formalism maintains the dependencies among the actions and the state variables in the composed model (as reflected in methods like `AffectedSVs` and `EnablingSVs`) based on the sharing approach and using the same dependencies in the underlying submodels. Currently, the Möbius tool features two composed model formalisms that use equivalence sharing: *Replicate/Join* [21, 39] and *Graph composition* [27, 39, 40]. We have also developed a prototype composition formalism to demonstrate action sharing.

B. Reward Models

Reward models [28] 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 the Möbius tool: a *performance variable* (PV). A PV[1] 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 PV can be specified to be measured at an instant of time, to be measured in steady state, to be accumulated over a period of time, or to be time-averaged over a 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 for the probability of the measure falling within a specified range.

C. Studies

During the specification of atomic, composed, and reward models in the tool, *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 specific value.

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*. The Möbius tool supports several study editors, the most sophisticated of which is based on a Design of Experiments approach (DOE [41]). 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

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

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.

D. Solution Techniques

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 operate on models only through the Möbius AFI. Formalism-specific analysis and property discovery is performed by the formalism editor. Möbius allows formalism-specific solution techniques in the form of property-specific solvers. For example, analytic solution using state-space generation requires the properties described above. In order to evaluate the overhead resulting from the generality of the Möbius framework, we have compared the performance of the simulators in *UltraSAN* and Möbius [42]. We observed that the amount of overhead is negligible compared to runtime differences caused by other factors, such as the algorithm used by the solver and the optimization techniques used in the implementation.

D.1 State-Space Generator

The Möbius tool also supports a variety of analytical/numerical solvers. The first step in analytic solution with the Möbius tool is the generation of a state space, done by the 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 of several implemented analytical/numerical methods may be employed to solve for the required performance variables. Figure 10 presents pseudo-code for the Möbius state-space generator, with AFI method calls highlighted.

D.2 Simulation

The Möbius tool currently supports two modes of discrete event simulation: transient and steady-state. In the transient mode, the simulator uses the independent replication technique to obtain statistical information about the specified reward variables. In the steady-state mode, the 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.

```
STATE-SPACE GENERATOR(Model)
  AllStates ← ∅
  States ← {Model.CurrentState()}
  Actions ← Model.ListActions()
  while (States ≠ ∅) do
    S ← States.Pop()
    Model.SetState(S)
    Enabled ← ∅
    for (Act ∈ Actions) do
      if (Act.Enabled()) then
        Enabled ← Enabled ∪ {Act}
    while (Enabled ≠ ∅) do
      Act ← Enabled.Pop()
      Act.Fire()
      S′ ← Model.CurrentState()
      if (S′ ∉ AllStates) then
        States ← States ∪ {S′}
        AllStates ← AllStates ∪ {S′}
      Model.SetState(S)
  return AllStates
```

Fig. 10. Pseudo-code for the Möbius state-space generator

The simulator 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. We accomplish this parallelism by running different observations on different workstations in the case of transient simulation, or by running different trajectories in the case of batch means. We have observed that this level of parallelism usually yields near-linear speedup.

Like the code for the state-space generator, the code for the operation of the Möbius simulator can be written using only the AFI method calls mentioned in Tables II and III.

VII. Conclusion

Through the creation of the Möbius framework and implementation, we have created an environment in which multiple modeling formalisms and solvers can interact in a tightly integrated environment. Because of careful construction of the AFI, it is possible to add new formalisms and solvers into the tool with a minimum of disruption to existing components. This paper describes the framework and implementation, and illustrates their usefulness through a number of formalisms and examples.

To accomplish our goals, Möbius must be able to accommodate a variety of different formalisms, including the state variables and actions required by the formalisms. Möbius has a general and flexible state variable typing system that accommodates a large number of formalism state variables. The Möbius execution policy and actions generalize the execution policies of most formalisms. Consequently, we can show that mappings exist from formalisms to the Möbius framework. Möbius also has properties that preserve information that is important for efficient solution, but that may be lost in the translation.

The Möbius tool implements most of the framework's capabilities with the AFI. Models interact with other models and solvers through the use of the AFI, yielding a tightly integrated environment. We presented two atomic model for-

malisms that have been implemented in the tool: SANs and $PEPA_k$. We also showed how other parts of the tool, such as reward models, composed models, studies, and solvers, interact with atomic models without regard to the formalism that was used to create them.

New formalisms and solvers are currently under development by ourselves and by others. Others can benefit by leveraging existing formalisms and tools. For example, a formalism developer does not need to recreate a distributed simulation solver or a state-space generator, or composition formalisms, for the formalism to be useful. Our recent 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. We would also like to thank Jenny Applequist for her editorial assistance.

References

[1] 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*, vol. 22, no. 2, pp. 248–260, Apr. 1975.

[2] S. Donatelli, "Superposed generalized stochastic Petri nets: Definition and efficient solution," in *Application and Theory of Petri Nets 1994, LNCS 815 (Proc. 15th International Conference on Application and Theory of Petri Nets, Zaragoza, Spain)*, R. Valette, Ed., pp. 258–277. Springer-Verlag, June 1994.

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

[4] M. K. Molloy, "Performance analysis using stochastic Petri nets," *IEEE Trans. on Comp.*, vol. 31, pp. 913–917, Sept. 1982.

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

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

[7] 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*, Zaragoza, Spain, Sept. 1999, pp. 2–9.

[8] B. R. Haverkort, "Performability evaluation of fault-tolerant computer systems using DyQN-Tool$^+$," *International Journal of Reliability, Quality, and Safety Engineering*, vol. 2, no. 4, pp. 383–404, 1995.

[9] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo, "GreatSPN 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets," *Performance Evaluation*, vol. 24, no. 1–2, pp. 47–68, Nov. 1995.

[10] M. Ajmone Marsan, G. Balbo, and G. Conte, "A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems," *ACM Transactions on Computer Systems*, vol. 2, pp. 93–122, 1984.

[11] 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, Oct. 1995.

[12] 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)*, San Diego, California, Jan. 1993, pp. 390–391.

[13] G. Ciardo, A. Blakemore, P. F. J. Chimento, J. K. Muppala, and K. S. Trivedi, "Automated generation and analysis of Markov reward models using stochastic reward nets," in *Linear Algebra, Markov Chains, and Queueing Models*, C. Meyer and R. J. Plemmons, Eds., pp. 141–191. Heidelberg: Springer-Verlag, 1993.

[14] R. M. L. R. Carmo, L. R. de Carvalho, E. de Souza e Silva, M. C. Diniz, and R. R. Muntz, "TANGRAM-II," in *Computer Performance Evaluation: Modelling Techniques and Tools: Proceedings of the 9th International Conference*, Raymond Marie, Brigitte Plateau, Maria Calzarossa, and Gerardo Rubino, Eds., St. Malo, France, June 1997, pp. 6–18.

[15] R. J. Pooley, "The integrated modelling support environment: A new generation of performance modelling tools," in *Computer Performance Evaluation: Modelling Techniques and Tools: Proceedings of the 5th International Conference*, G. Balbo and G. Serazzi, Eds., Torino, Italy, Feb. 1991, pp. 1–15.

[16] 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)*, Singapore, Sept. 1997, pp. 471–476.

[17] Aad P. A. van Moorsel and Yiqing Huang, "Reusable software components for performability tools, and their utilization for web-based configuration tools," in *Computer Performance Evaluation: Modelling Techniques and Tools: Proceedings of the 10th International Conference*, R. Puigjaner, N. N. Savino, and B. Serra, Eds., Palma de Mallorca, Spain, Sept. 1998, pp. 37–50.

[18] 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.

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

[20] F. Bause, P. Buchholz, and P. Kemper, "A toolbox for functional and quantitative analysis of DEDS," in *Computer Performance Evaluation: Modelling Techniques and Tools: Proc. of the 10th Int. Conf.*, R. Puigjaner, N. N. Savino, and B. Serra, Eds., Palma de Mallorca, Spain, Sept. 1998, pp. 356–359.

[21] 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*, vol. 9, no. 1, pp. 25–36, Jan. 1991.

[22] G. Clark and W. H. Sanders, "Implementing a stochastic process algebra within the Möbius modeling framework," in *Process Algebra and Probabilistic Methods: Performance Modelling and Verification: Proc. of the Joint International Workshop, PAPM-PROBMIV 2001*, Aachen, Germany, September 2001, vol. 2165 of *Lecture Notes In Computer Science*, pp. 200–215, Berlin: Springer.

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

[24] D. D. Deavours and W. H. Sanders, "'On-the-fly' solution techniques for stochastic Petri nets and extensions," *IEEE Trans. on Soft. Eng.*, vol. 24, no. 10, pp. 889–902, Oct. 1998.

[25] D. D. Deavours and W. H. Sanders, "An efficient well-specified check," in *Proceedings of the 8th International Workshop on Petri Nets and Performance Models (PNPM '99)*, Zaragoza, Spain, Sept. 1999, pp. 124–133.

[26] W. D. Obal II and W. H. Sanders, "State-space support for path-based reward variables," in *Proc. 3rd Ann. IEEE International Computer Performance and Dependability Symposium (IPDS '98)*, Durham, North Carolina, Sept. 1998, pp. 228–237.

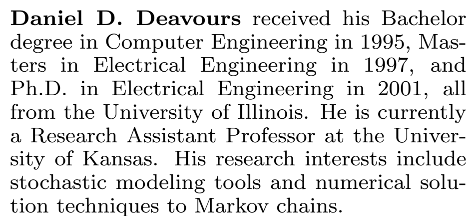
[27] W. D.. Obal II and W. H. Sanders, "Measure-adaptive state-space construction methods," *Performance Evaluation*, vol. 44, pp. 237–258, Apr. 2001.

[28] 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., vol. 4 of *Dependable Computing and Fault-Tolerant Systems*, pp. 215–237. Heidelberg: Springer-Verlag, 1991.

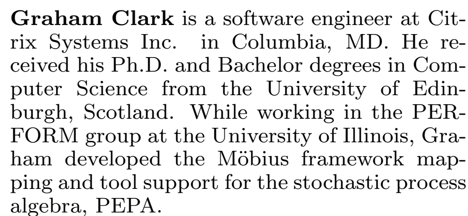
[29] B. Plateau and K. Atif, "A methodology for solving Markov models of parallel systems," *IEEE Journal on Software Engineering*, vol. 17, no. 10, pp. 1093–1108, 1991.

[30] P. Kemper, "Numerical analysis of superposed GSPNs," in *Sixth*

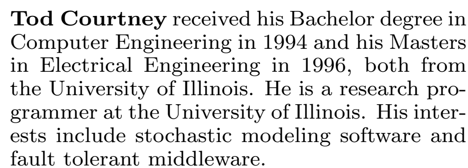
International Workshop on Petri Nets and Performance Models (PNPM '95), Durham, North Carolina, Oct. 1995, pp. 52–61.
[31] G. Ciardo and K. S. Trivedi, "A decomposition approach for stochastic reward net models," *Performance Evaluation*, vol. 18, pp. 37–59, 1993.
[32] J. M. Doyle, "Abstract model specification using the Möbius modeling tool," M.S. thesis, University of Illinois at Urbana-Champaign, Jan. 2000.
[33] C. H. Sauer and Edward A. MacNair, *Simulation of Computer Communication Systems*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1983.
[34] A. Bobbio, A. Puliafito, and M. Telek, "A modeling framework to implement preemption policies in non-Markovian SPNs," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 36–54, Jan. 2000.
[35] 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, pp. 135–144.
[36] Daniel D. Deavours, *Formal Specification of the Möbius Modeling Framework*, Ph.D. thesis, University of Illinois at Urbana-Champaign, Oct. 2001.
[37] D. R. W. Holton, "A PEPA specification of an industrial production cell," *The Computer Journal*, vol. 38, no. 7, pp. 542–551, 1995.
[38] S. Gilmore, J. Hillston, and M. Ribaudo, "An efficient algorithm for aggregating PEPA models," *IEEE Transactions on Software Engineering*, vol. 27, no. 5, pp. 449–464, 2001.
[39] 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, pp. 241–250.
[40] A. J. Stillman, "Model composition within the Möbius modeling framework," M.S. thesis, University of Illinois at Urbana-Champaign, 1999.
[41] D. Montgomery, *Design and Analysis of Experiments*, John Wiley & Sons, Inc., 5th edition, 2001.
[42] A. Williamson, "Discrete event simulation in the Möbius modeling framework," M.S. thesis, University of Illinois at Urbana-Champaign, 1998.



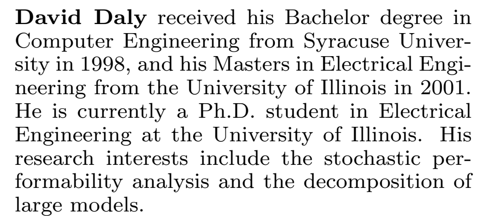
Daniel D. Deavours received his Bachelor degree in Computer Engineering in 1995, Masters in Electrical Engineering in 1997, and Ph.D. in Electrical Engineering in 2001, all from the University of Illinois. He is currently a Research Assistant Professor at the University of Kansas. His research interests include stochastic modeling tools and numerical solution techniques to Markov chains.



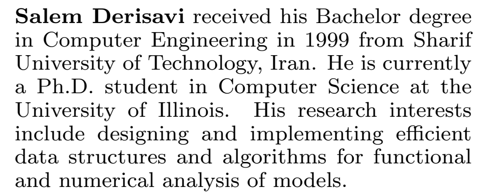
Graham Clark is a software engineer at Citrix Systems Inc. in Columbia, MD. He received his Ph.D. and Bachelor degrees in Computer Science from the University of Edinburgh, Scotland. While working in the PERFORM group at the University of Illinois, Graham developed the Möbius framework mapping and tool support for the stochastic process algebra, PEPA.



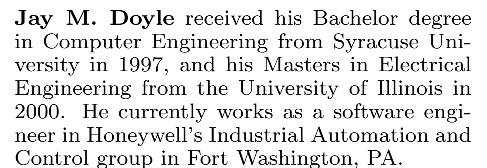
Tod Courtney received his Bachelor degree in Computer Engineering in 1994 and his Masters in Electrical Engineering in 1996, both from the University of Illinois. He is a research programmer at the University of Illinois. His interests include stochastic modeling software and fault tolerant middleware.



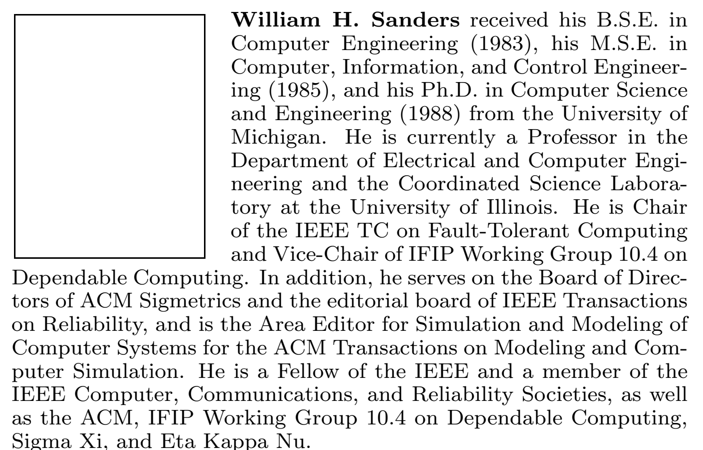
David Daly received his Bachelor degree in Computer Engineering from Syracuse University in 1998, and his Masters in Electrical Engineering from the University of Illinois in 2001. He is currently a Ph.D. student in Electrical Engineering at the University of Illinois. His research interests include the stochastic performability analysis and the decomposition of large models.



Salem Derisavi received his Bachelor degree in Computer Engineering in 1999 from Sharif University of Technology, Iran. He is currently a Ph.D. student in Computer Science at the University of Illinois. His research interests include designing and implementing efficient data structures and algorithms for functional and numerical analysis of models.

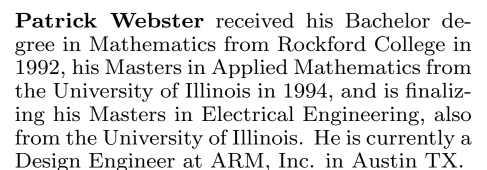


Jay M. Doyle received his Bachelor degree in Computer Engineering from Syracuse University in 1997, and his Masters in Electrical Engineering from the University of Illinois in 2000. He currently works as a software engineer in Honeywell's Industrial Automation and Control group in Fort Washington, PA.



William H. Sanders received his B.S.E. in Computer Engineering (1983), his M.S.E. in Computer, Information, and Control Engineering (1985), and his Ph.D. in Computer Science and Engineering (1988) from the University of Michigan. He is currently a Professor in the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory at the University of Illinois. He is Chair of the IEEE TC on Fault-Tolerant Computing and Vice-Chair of IFIP Working Group 10.4 on Dependable Computing. In addition, he serves on the Board of Directors of ACM Sigmetrics and the editorial board of IEEE Transactions on Reliability, and is the Area Editor for Simulation and Modeling of Computer Systems for the ACM Transactions on Modeling and Computer Simulation. He is a Fellow of the IEEE and a member of the IEEE Computer, Communications, and Reliability Societies, as well as the ACM, IFIP Working Group 10.4 on Dependable Computing, Sigma Xi, and Eta Kappa Nu.

Dr. Sanders's research interests include performance/dependability evaluation, dependable computing, and reliable distributed systems. He has published more than 100 technical papers in those areas.



Patrick Webster received his Bachelor degree in Mathematics from Rockford College in 1992, his Masters in Applied Mathematics from the University of Illinois in 1994, and is finalizing his Masters in Electrical Engineering, also from the University of Illinois. He is currently a Design Engineer at ARM, Inc. in Austin TX.