The Möbius Execution Policy [*]

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

Möbius is an extensible framework and tool for performance and dependability modeling, and supports multiple modeling formalisms and solvers. As a framework, Möbius must be sufficiently general to capture the various formalism behaviors. Specifically, the Möbius execution policy must be flexible enough to accommodate the execution policies of all formalisms implemented in the framework. We know of no existing execution policy which is capable of doing this and meeting the many other goals of the framework. We present the Möbius execution policy that addresses these needs. In developing the policy, we have generalized the various preemption policies and made all aspects of the execution policy state-dependent, which has never before been considered. Because all aspects may be state-dependent, we also had to relax the assumption that work proceeds at a constant rate; this is also novel. Finally, we show that within the context of Möbius, the extra structure and overhead needed to implement a particular behavior can largely be avoided except when that behavior is present in a model.

1 Introduction

Möbius [6, 7, 13] is an extensible multi-formalism multi-solution performance and dependability modeling framework for modeling discrete event systems. Möbius allows models to be expressed in a variety of different formalisms, and allows models to be solved with a variety of different solvers. Some parts of a model may be described in one formalism, and other parts in another. The parts may be composed using a third, "composition" formalism. An example of a composition formalism is the formalism, such as SGSPNs superposed generalized stocahstic Petri nets (SGSPNs) [8], where submodels are synchronized on transitions. In the Möbius framework, any solver may be used to solve a model so long as the requirements for that solver are met. Thus, models expressed in different formalisms may interact, and solvers may be applied (whenever model properties allow) independent of the formalisms used to express a model. New formalisms and solvers may also be added with a minimum of change to existing formalisms and solvers.

An important aspect of any modeling formalism is the definition of its execution policy. An *execution policy* is a set of rules for unambiguously defining the underlying stochastic process of a model. Defining an execution policy for a formalism involves specifying when certain state-change events in a model can occur, and what happens when they occur. Execution policies have been studied for some time. Examples of execution policies include various preemption policies such as preemptive resume (prs, also called race with enabling memory), preemptive repeat different (prd), and preemptive repeat identical (pri) [4], as well as reactivation [12], which we review in detail below.

While each of these execution policies is suitable for representing behaviors within the formalism for which it was intended, it is not suitable for a general modeling framework such as Möbius. In particular, the multi-formalism, multi-solution nature of the Möbius framework makes it necessary to develop a execution policy that is flexible, and can be tuned to represent a wide variety of behaviors. This requirement necessitated research in several areas that have not been examined in previously existing research, such as how to integrate the concept of reactivation with the prs and pri policies. Furthermore, the Möbius execution policy must support the specification of a variety of state-dependent be-

[*] 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.

haviors, and many existing execution policies do not support this generality. It is therefore important to develop a new execution policy, designed for models expressed with the Möbius framework, that can support the wide variety of behaviors expressible by formalisms within the framework.

The execution policy we developed for the Möbius framework is not simply a union of the existing execution policies; it extends the concepts presented by various execution policies. In doing so, it generalizes the various preemption policies and allows them to be completely state-dependent. It allows for completely state-dependent delay distributions. Möbius also provides a general mechanism for converting the concept of age between different delay distributions. To accomplish these things, Möbius uses an "action" to represent the various formalism state-change mechanisms. The state of an action can be captured using five variables. The execution policy is defined by a few simple rules. It is possible to express the various policies by making three independent decisions, resulting in eight possibilities. These possibilities capture and extend the various execution policy behaviors in a state-dependent way.

As we have just argued, the execution policy we have developed for the Möbius framework is more general than that defined for most formalisms, and hence can represent behaviors that have not previously been considered. For this reason, it not only solves the practical problem we face in Möbius, but also may be of interest to the larger community. It is structured, regular, and coherent. It allows all aspects of the execution policy to be state-dependent, allows for arbitrary changes in delay distribution, and does both of these things in a consistent way. We believe that the resulting execution policy is elegant and is simpler than it would have been if it had been based on the union of a number of different execution policies.

In this paper, we begin in Section 2 by reviewing previous work, concerning both Möbius and other execution policies, and include a motivating example. Section 3 then shows the details of the our solution for Möbius in the action state and execution policy rules. In Section 4, we derive the equations needed to describe the behavior of an action, and discuss issues with solvers in Section 5. We conclude in Section 6.

2 Motivation

2.1 Möbius framework description

We refer to [7, 13] for a more detailed description of the Möbius framework. For our discussions, it is sufficient to know that Möbius makes it possible to use multiple formalisms to describe a single model. This means, for example, that a portion of a model may be described using one formalism, such as generalized stochastic Petri nets (GSPNs) [2], while another part of the model is described with a different formalism, such as PEPA [11]. Submodels of different formalisms may be combined into a single, larger model using a technique called model composition, e.g., Replicate/Join [14], or SGSPNs.

2.2 Execution policies

One of the major challenges in developing the Möbius execution policy was to find an execution policy that encompasses all the policies used in various formalisms. We begin with a brief survey of the major existing execution policies we considered in developing the Möbius execution policy.

For consistency, we use the term *action* for the basic state-change mechanism of all formalisms. The most commonly used policy is prd [4, 1]. With prd, an action will complete in some random time after it becomes enabled. If, in the interval between enabling and completion, it becomes disabled or otherwise interrupted, then when the action becomes enabled again, it must choose a new completion time and essentially start over.

With prs, if an action becomes disabled before it completes, then the action is suspended. When the action becomes enabled again, it may resume, that is, continue as if it had not been disabled.

In a third policy, pri [3], when an action becomes disabled before it completes, the process must start over when it becomes enabled again, but it keeps the same completion time. Consequently, if the action is enabled for t time before becoming disabled, the action must be enabled for at least t time before there is any possibility that the action may complete.

While these three policies are commonly used, other interesting extensions have been proposed. For example, SANs [12] include the concept of reactivation. An action reactivates when the model reaches some particular state or states, and the action is enabled. When reactivation occurs, the action must start over and choose a new completion time. This is the same thing as prd when an action becomes disabled, and is essentially the equivalent of a "reset button."

GSMPs [10, 5] are commonly used to describe a class of simulation languages in which the inter-event distribution may be generally distributed, but the next state is only dependent on the current state. Taking this as the definition, our approach is within the GSMP class, and can express GSMP models. However, we add structure and methodology. Just as the prs and pri policies described in [4] use an age and resample variable, we have additional variables, and with these variables we are able to address complex and useful behaviors systematically.

2.3 Generalizing execution policies

We are presented with several challenges in developing the Möbius execution policy. On one hand, we have several preemption policies, such as prs, prd, and pri [4]. As defined in [4], these preemption policies are static in that the policies do not change as a model changes state, and the delay characteristics of an action also do not change as a model changes state. Elsewhere, e.g., [9], delays that are allowed to change only by a constant scaling factor have been considered, but the policies are still not allowed to change as a function of model state.

On the other hand, many other formalisms, e.g., SANs [12], allow the delays of actions to change arbitrarily as a function of the model state. Early work on this generalization led to a problem of defining on *which* state the delay should depend: the state in which the action becomes enabled, the state in which it completes, or perhaps some state in the interim. This led to the concept of "reactivation," which requires the action to "start over" as an aid to controlling state-dependent behavior.

Integrating these various execution policies is necessary because of model composition. Since a Möbius model may be made up of submodels of different formalisms, each of which has different execution policies, Möbius must be able to accommodate each formalism execution policy. This is challenging because many aspects of these policies have not been considered in combination with others. Since many formalisms implement general state-dependent behavior, Möbius must accommodate this behavior. However, if we allow general state-dependent delays, we have no mechanism for providing prd and pri. Furthermore, a composition formalism may synchronize a GSPN transition with a pri policy and a SAN activity; how can one accommodate both pri and reactivation?

One possibility would be to take a union of all the existing execution policies of formalisms and use that union as the Möbius execution policy. This is an adequate solution, but it has several problems. First, it precludes the possibility of synchronizing the GSPN transition and the SAN activity. Another problem is that it restricts the possible expressible behaviors. If, for example, a new execution policy is discovered, or some innovations cause some restriction on an existing policy to be relaxed, Möbius would have no way of accommodating this, and hence would not be extensible.

An ideal solution would be to find a generalization of the various execution policies. This generalization would allow for complete state dependence. For example, an action could change not only its distribution, but its execution policy. Further, it would be able to capture both pri and reactivation, allowing for synchronization of a transition and activity. We note that many of the restrictions placed on the execution policies exist so that analytic solution is possible or tractable. Möbius has properties [7], which is able to capture these restrictions, so a generalized execution policy does not preclude efficient solution.

We propose such a generalized execution policy. A consequence of this is the relaxation of what we call the "constant work assumption." For example, consider the formulation of the generalized semi-Markov process, or GSMP, given in [5, 10]. Once an event is enabled, the "event rate" can change only by a constant as a function of model state; that is, the delay distribution can change only by a linear scaling. What is specifically absent is the ability for the delay to change in an arbitrary way while the event is still enabled. Adding this feature does require some additional structure. However, we believe that the resulting execution policy is more regular, simpler, and more extensible than one based on a union of a number of different execution policies.

2.4 Motivating example

This example illustrates not only the specific needs of the Möbius execution policy, but also the deficiencies of current execution policies in capturing interesting behaviors. The example is somewhat simplistic for the sake of conciseness, but is based on reasonable behaviors.

Consider a system that has failed due to the failure of one component. Three repairpeople, using three different but simplistic approaches, may perform the repair. Each repairperson keeps notes of whatever he or she has done.

The first repairperson performs a complex diagnostic process that takes a fixed amount of time, but is certain to find the failed component. At the beginning of the process, all components are suspect and belong to a suspect set. As the diagnostic process continues, certain components are eliminated from the suspect set. Eventually, in a predictable, fixed amount of time, the first repairperson will deduce which component has failed and be able to replace it. Components are eliminated from the suspect set roughly linearly with time. Let c be the time to repair. Thus, the distribution function describing the time between enanbing and completion of the action, called Delay, is defined so that $\mathrm{Delay}_1(t) = 0$ for $t < c$ and 1 for $t \geq c$. Note that since components are eliminated from the suspect set at a constant rate, we can say that in some sense, the rate at which work is being performed is constant.

The second repairperson uses a random algorithm for performing the repair of the system. He randomly chooses a component and replaces it with a component known to work. If the system becomes operational, the repair is complete; if it doesn't, he replaces the original component and randomly selects a new component (without checking whether he has already tested it). Assuming a large number of components and some small variance among replacement times, we can approximate the time to repair as an exponential random variable (as opposed to a geometric). Thus, $\mathrm{Delay}_2(t) = 1 - e^{-\lambda t}$. Note that in some sense the efficiency of the repair process decreases over time. At the beginning, the repairperson is unlikely to select a component that has already been selected, but as time progresses, the chances increase. Thus, the "work" performed by the repairperson decreases (exponentially) over time.

A third repairperson uses a more systematic approach. She also randomly chooses components to swap, but she only chooses components that have not been eliminated from the suspect set, either by diagnostics or previous swapping. (We assume that the time she takes to check her notes is negligible compared to the time needed to perform the replacement.) Again, we can approximate the time to repair as a uniform random variable. Thus, $\mathrm{Delay}_3(t) = t/b$ for $0 \le t \le b$. Note that for this repairperson, the efficiency of the repair process does not decrease over time, but remains constant. Thus, the "work" performed by this third repairperson is constant.

Any of these repair processes may be modeled using any formalism in which the delays may be generally distributed. However, consider the following scenario. Repairperson 1 works on the repair for some time t_1 and stops. Repairperson 2 then works on the repair for some time t_2 and stops. Finally, repairperson 3 starts working on the repair. She uses the information about the diagnostics performed by repairperson 1 and the components swapped by repairperson 2. How long does it take her to complete the repair, or, stated differently, what is the residual delay for repairperson 3? More importantly, how can we express this problem?

The difficulty in expressing this behavior using any existing formalism is this: the amount of time it takes repairperson 1 to perform e work is *not linear* in the amount of time it takes repairperson 2 to perform the same amount of work. Thus, there is no simple scaling relationship of the repair times, or the residual times, so the linear scaling approaches taken by GSMPs, for example, are inadequate. We also do not know of any way to express this behavior using multiple GSMP events (or stochastic Petri net (SPN) transitions).

One solution might be to allow the user to express the behavior in terms of the simulation "clocks." While accessing clocks directly is technically outside the scope of GSMPs, some simulators allow users to do this. The primary difficulty with this within the context of Möbius is that expressing any interesting behavior presupposes simulation as the solution technique. As shown in [9], a number of interesting non-Markovian execution policies can be solved analytically, and these solution techniques may be eliminated from the scope of Möbius if we express non-trivial execution policies in terms of simulation clocks.

We need a more structured way to express these interesting behaviors, not in terms of simulation clocks, but in terms of useful descriptions of general behaviors. That would yield an approach to describing the execution policy that is more disciplined and structured, yet general enough to capture all the previously described execution policies.

3 Realization of Möbius execution policy

3.1 Process paradigm

We begin by building a simple, intuitive terminology for describing behaviors that we wish to model. We use the term "process" for the basic unit of behavior that corresponds to the basic state-changing mechanism of a formalism. We say that a *process* is a *worker* performing *work* on a *task*. Thus, we appeal to the basic language construct of a subject and a verb, plus some criterion for completion, to describe behaviors. This serves as the basis of our conceptual model. An example of a process may be a program execution, where the worker is the CPUs, the work is the execution of machine instructions, and the task is the performance of some computation. Similarly, a repair process may be a repairperson (worker) performing diagnosis and repair (work) on a system until the system is made operational (task).

A process exists in an environment and interacts with that environment. The process can affect the environment in one simple way: when the process completes, the state of the system is changed to reflect the completion of the process. For example, when the computer completes execution of the program, the number of jobs in the system may be decreased by one, or when a repair is completed, the system may become operational.

The environment can affect a process in a number of ways. The most simple way is by allowing or disallowing the performance of work. This corresponds to the enabling or disabling of an action. The environment can change the degree of favorability of the work; in particular, the environment may be more or less favorable, and thus the worker may perform work at a faster or slower rate. This corresponds to state-dependent rates, for which the delay can change by a constant scaling factor. The constant work assumption holds if these ways are the only ways the environment can affect a process. For example, a deep space probe may require a reduction in power consumption, so the microcontroller may operate at a slower clock frequency. The delay and work characteristics are changed by a constant scaling factor.

A more interesting way the environment may affect a process is by causing the worker to change. This is the case in our motivating example. Different repairpeople with different delay and work characteristics are operating on the same task.

The environment may change and cause the task to change as well. Sometimes an action is associated with the worker, sometimes with the task, and sometimes with the pair. Although less frequently used, our approach addresses the possibility that an action is associated with a worker and that the task assigned to the worker changes. This is useful when the work performed by the first worker may be transferred to the new task. That may be the case, for example, when a real-time program misses a deadline. The program, having missed the deadline, may work on a more difficult problem to which the previous computation may be applied.

Changes in either the worker or the task can affect the characteristics of the delay, and the changes may be more complex than scaling factors. As the example in Section 2.4 illustrates, the delay may change from a deterministic to an exponential to a uniform random variable, and the work characteristics may change as well. In order to model this kind of behavior, we must develop mechanisms to describe this kind of complex behavior precisely. To do this, we define the notion of an event.

3.2 Events

An event is a state change that occurs in a model. Formally, an *event* is the 4-tuple $(\sigma, \tau, a, \sigma')$: the current state, the sojourn time, the action that completes, and the resulting next state. Note that the current state and action completion uniquely define the next state. (We include the next state because model composition may change what the unique next state may be.)

In order to describe our execution policy, we must categorize events, because different behaviors may occur for different types of events. Each event is categorized with respect to a particular action a_0.

Enabling event: any event in which an action becomes enabled, i.e., for $(\sigma, \tau, a, \sigma')$, a_0 is not enabled in σ but is enabled in σ'.

Disabling event: any event in which an action becomes disabled, i.e., for $(\sigma, \tau, a, \sigma')$, a_0 is enabled in σ but not enabled in σ', and $a_0 \neq a$.

Completing event: any event in which an action completes, i.e., for $(\sigma, \tau, a, \sigma')$, $a_0 = a_i$.

Interrupting event: any event in which the action remains enabled may be an interrupting event. An interrupting event causes an action to behave much as it would if it encountered a disabling event followed immediately by an enabling event. I.e., any event $(\sigma, \tau, a, \sigma')$ in which a_0 is enabled in σ and σ' and $a_0 \neq a$ may be an interrupting event.

Defining event: any enabling, disabling, or interrupting event.

Finally, we use a prime to distinguish the action state after an event occurs. For example, Start and Start' are the action state, before and after an event, respectively.

3.3 Action state

Recall that an action is the basic state-change mechanism of the Möbius model. An action is composed of action functions, which describe how an action behaves and are discussed in [7], and action state, which captures the current state of an action. The action state is made up of five action state variables.

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

We begin by describing the action state that we use. Then we compare and contrast our choice of action state with those of other formalisms, and explain and justify the differences.

$\mathrm{Start} : \mathbb{R}$ The most recent time that the associated action became enabled or was interrupted.

$\mathrm{Delay} : \mathbb{R} \to [0,1]$ The delay distribution function is defined such that $\mathrm{Delay}(t)$ is the probability that the associated action will complete by time t.

$\mathrm{Effort} : \mathbb{R} \to [0,1]$ The effort function is defined such that $\mathrm{Effort}(t)$ is the amount of work performed in t time. We require it to have certain properties that are identical to the properties of a valid cumulative distribution function. These are straightforward: $\mathrm{Effort}(0-) = 0$ (to enforce causality), $\mathrm{Effort}(\infty) = 1$, Effort is nondecreasing, and Effort is right-continuous. Since both the delay distribution and effort functions describe how the action evolves over time, the two functions are always tied together. For example, if the environment changes the degree of favorability, both the delay distribution and effort functions will change. Note that work is unitless.

$\mathrm{WE} : [0,1]$ The worker effort is a measure of the amount of work the worker has performed on the task, and is computed by evaluating the effort function. For example, in our repairperson example, if the first repairperson has eliminated half of the components from the suspect set in t_1 time, then $\mathrm{WE} = \mathrm{Effort}(t_1) = 0.5$.

$\mathrm{MTE} : [0,1]$ The minimum task effort is the minimum effort known to be required by the task. While the worker effort is the amount of work performed by the worker, the minimum task effort is the minimum amount of work that the worker must perform in order to complete the task. This is the concept used in pri, for example. Note the invariant $\mathrm{WE} \leq \mathrm{MTE}$.

These five variables are sufficient to capture the state of an action. They describe the delay characteristics, the work characteristics, how much work has already been performed, and how much we know is needed for the action to complete. Next, we give the rules for how these states are manipulated as the model evolves.

3.4 Rules for action state

The rules for the start time are simple. The start time represents the time of the most recent enabling or interrupting event for the associated action. It holds valid information only when an action is enabled. When an action completes, all the variables are set to 0 or the empty set.

At any defining event, the delay, effort, worker effort, and minimum task effort may be preserved or discarded. This means different things for each variable. If the delay and effort function are preserved, then $\mathrm{Delay}' = \mathrm{Delay}$ and $\mathrm{Effort}' = \mathrm{Effort}$. If they are both discarded, then $\mathrm{Delay}' = \emptyset$ and $\mathrm{Effort}' = \emptyset$. By using separate action state variables for effort and delay, we avoid any ambiguity due to state-dependent delay distributions. In addition, the definition and use of interruptions allows for precise control in indicating the state on which any state-dependent behavior may depend.

If the worker effort is preserved and the defining event is an enabling event, then $\mathrm{WE}' = \mathrm{WE}$. If the defining event is an interrupting or disabling event, then the computation is more difficult. Let t_0 be the amount of time the action would need to be enabled to perform WE effort, i.e., $t_0 = \mathrm{Effort}^{-1}(\mathrm{WE})$. Let τ be the current time minus Start. Then $\mathrm{WE}' = \mathrm{Effort}(t_0 + \tau)$. This essentially says that the new worker effort is the old worker effort plus the effort performed since the last enabling or interrupting event. If the worker effort is discarded, then $\mathrm{WE}' = 0$. Our repair example in Section 2.4 is an example of a situation in which we would want to preserve the worker effort. This also occurs with the prs preemption policy.

If the minimum task effort is preserved, then MTE' is set to the greater of MTE and WE'. Following this rule maintains the desired property that $\mathrm{WE} \leq \mathrm{MTE}$, and simply states that the amount of work known to be required is at least as large as the amount of work already applied. If the minimum task effort is discarded, then $\mathrm{MTE}' = 0$.

3.5 Enumeration of preserve and discard possibilities

In the previous section, we introduced the five variables used to capture the behavior of a process. Note that for defining events, there are three independent decisions that must be made: preserve or discard WE, preserve or discard MTE, and preserve or discard Delay and Effort functions. This results in eight possibilities, each of which we discuss in turn. We label each of them with three letters, each a P or a D. The first letter indicates whether we preserve or discard the worker effort, the second indicates whether we preserve or discard the minimum task effort, and the third indicates whether we preserve or discard the delay distribution and effort functions.

Note that in our framework, at a defining event we can only make a binary decision: preserve or discard. We do not allow for partial discarding. This is an intended limitation of our approach. However, more complex behaviors that may include partial discarding may be expressed using several actions and state variables.

PPP: preserve WE, preserve MTE, preserve Delay and Effort. This is the same as the prs of queueing formalisms or stochastic Petri nets. At a defining event, all the variables are preserved so that when the action is enabled, it may continue where it left off. This would be the case, for example, when a computer program has been suspended and later resumes execution, and the program execution continues where it left off.

PPD: preserve WE, preserve MTE, discard Delay and Effort. This is a significant generalization of the PPP or prs case, because it allows distributions to change at defining events. Here, the action continues where it left off, but it continues with (potentially) different delay and work characteristics. This is a condition that other approaches fail to consider adequately. The need for PPD is illustrated in our example in Section 2.4.

PDP: preserve WE, discard MTE, preserve Delay and Effort. This is an interesting and unusual case in which everything but the minimum task effort is preserved. It corresponds to a situation in which the action resumes, but the information about the minimum task effort is lost. This behavior occurs infrequently, but it is important to be able to model it accurately when it does.

PDD: preserve WE, discard MTE, discard Delay and Effort. This is a generalization of the previous PDP case. As with PDP, this behavior occurs infrequently, but the ability to represent it is useful.

DPP: discard WE, preserve MTE, preserve Delay and Effort. This is the same as the pri policy that is discussed in [3]. The action must start over, but keeps the same completion time.

DPD: discard WE, preserve MTE, discard Delay and Effort. This is a generalization of the DPP or pri case. Here, the worker must start over on the task, but new delay and work characterizations may apply. Unless the new delay and work characteristics are changed only by a scaling factor, existing methods are unable to express this possibility.

DDP: discard WE, discard MTE, preserve Delay and Effort. This is similar to the prd preemption policy and is perhaps the policy most frequently used in modeling. The delay and effort functions are preserved across multiple instances of being enabled as long as the action does not complete. If a formalism does not allow state-dependent behavior, then this case is identical to the DDD case.

DDD: discard WE, discard MTE, discard Delay and Effort. This is similar to the frequently used prd policy and DDP. If a formalism uses state-dependent delays, the state-dependent information is lost across defining events.

The clear distinction between DDP and DDD illustrates how the Möbius execution policy manages state-dependent behavior explicitly and consistently.

3.6 Justification for variables

Due to space limitations, we can offer only informal arguments to justify the selection and use of these variables to represent the state of an action. First, we note that the GSMP formalism [10, 5] uses only a clock structure to keep track of the residual delay for each action. This is equivalent to our use of the start variable and the delay distribution function.

The work of [4] uses three variables: the firing time, the age variable, and the resample indicator variable. The difference between the firing time and age variables is roughly equivalent to a clock in GSMPs. However, there is more flexibility with using an age variable and firing time than with clocks. For prs, the age variable and firing time are roughly equivalent to the Möbius worker effort, start time, and delay distribution. For pri, the age variable and firing time are roughly equivalent to the minimum task effort and delay distribution. The Möbius execution policy allows an action to exhibit both prs-like and pri-like behavior. It is to achieve this flexibility that we use three real-valued variables (Start, WE, and MTE) instead of two (age variable and firing time).

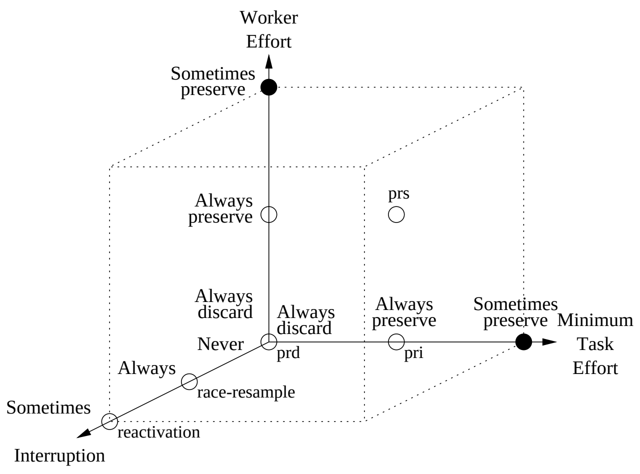

Figure 1. Möbius policy space.

Furthermore, since Möbius allows for general state-dependent behavior, we store the delay and effort functions as variables to allow the user to determine on which state any state-dependent behavior depends. The use of an effort function is also necessary in order to translate the worker effort and minimum task effort between different delay characteristics when there is no simple linear translation available. We argue that our relatively large number of variables is necessary in order to express the full generality of behaviors that we may encounter in the context of modeling formalisms implemented in the Möbius framework.

We illustrate the modeling expressiveness possible with the Möbius execution policy in Figure 1. The origin is the "simplest" case (prd), in which a disabled action has no state. The "Worker Effort" axis illustrates a qualitative increase in generality, with selective preservation of worker effort being the most general. Similarly, the "Minimum Task Effort" and "Interruption" axes increase from never to always to sometimes. The open circles illustrate points in the space that are implemented in existing execution policies. The Möbius execution policy allows an action to operate anywhere within the space. What the figure fails to capture is that our execution policy allows for complete state-dependent behavior, including relaxation of the constant work assumption.

4 Derivation of action delay characteristics

In this section, we describe how to calculate the delay characteristics of an action, given the action state described in the previous section.

4.1 Unique inverse of Effort

Our derivations require the inverse of the effort function. There is no guarantee that a unique inverse exists, because the effort function is non-decreasing. We need a unique inverse to compute the delay characteristics of an action.

For some time t, let $T_e = \{t : e = \mathrm{Effort}(t)\}$ be the set of elements that satisfy the inverse. We choose the unique inverse to be the smallest time in T_e, that is, $t = \mathrm{Effort}^{-1}(e) = \min T_e$. Finding a unique inverse is only a problem when the effort function is non-increasing for some interval. We chose the smallest time to be the unique inverse for two reasons. First, if an action is interrupted while the effort function is in a non-increasing interval, then we interpret this to mean that no useful work is being performed in that interval. If the action resumes, then it must start over at the beginning of the interval. Another reason we chose the smallest time to be the unique inverse is that if we chose some other time, there is the possibility that when the action resumes, it could "skip over" some time and possibly some probability mass, which seems illogical.

4.2 Effective delay distribution

Here, we derive the delay distribution of an action, taking into account its state. To do this, we use the unique inverse of the effort function discussed above. We call this the *effective delay distribution* or the *conditional delay distribution* because it is the delay distribution conditioned on the values of WE and MTE.

We can compute the effective delay distribution for an action at any defining event. Let e_w be the worker effort, e_m be the minimum task effort, and Delay and Effort be the distribution and effort functions for the action. First, we note that since $\mathrm{WE} \le \mathrm{MTE}$, $e_w \le e_m$. Let $t_w = \mathrm{Effort}^{-1}(e_w)$ and $t_m = \mathrm{Effort}^{-1}(e_m)$. We know that because Effort is nondecreasing, $t_w \le t_m$.

Let X be a random variable with the distribution function Delay. Let the action have a worker effort e_w and minimum task effort e_m, and let $t_0 = \mathrm{Start}$ be the time of the last enabling or interrupting event. Let t_w be the *effective worker time*, which we can compute using the unique inverse $t_w = \mathrm{Effort}^{-1}(e_w)$, and t_m be the *effective minimum task time*, again computed as $t_m = \mathrm{Effort}^{-1}(e_m)$. The effective delay distribution, $d(t_0 + t)$, gives the probability that the action will complete by time $t_0 + t$.

The information that we have is that the action has already performed work effectively for t_w time, and that it must run t_m time for there to be any chance the action will complete. We can write this precisely.

$$\begin{aligned} d(t_0+t) &= \Pr[X \le t + t_w | X > t_m] \\ &= \frac{\Pr[t_m < X \le t + t_w]}{1 - \Pr[X \le t_m]} \\ &= \begin{cases} 0 & : \ t + t_w \le t_m, \\ \frac{\mathrm{Delay}(t+t_w) - \mathrm{Delay}(t_m)}{1 - \mathrm{Delay}(t_m)} & : \ \text{otherwise.} \end{cases} \end{aligned}$$

Note that the equation has two regions. First, for time $t \in [0, t_m - t_w]$, the probability of the action completing is zero because the action has not been enabled for the minimum task time. In the second region, in which $t > t_m - t_w$, the action has performed more work than the minimum task effort. In this region, the conditional distribution is like the original distribution shifted to the right by t_w and scaled so that the remaining probability mass is scaled appropriately.

4.3 Solution to example

Using this solution, we are now able to solve the problem we posed in Section 2.4. Note that we implement the PPD policy each time the repairperson begins or ends working. The delay and effort function is determined by the identity of the repairperson, which is presumably indicated through a state variable.

Let $\mathrm{Effort}_1(t) = t/c$ for the first repairperson, $\mathrm{Effort}_2(t) = 1 - e^{-\lambda t}$ for the second repairperson, and $\mathrm{Effort}_3(t) = t/b$ for the third. The amount of effort performed by the first repairperson is calculated by $e_1 = \mathrm{Effort}_1(t_1)$, which is used for both WE and MTE. The amount of time the second repairperson would need to work in order to perform e_1 work is given by $t_{1e} = \mathrm{Effort}_2^{-1}(e_1)$. The amount of work performed by the second repairperson is then $\mathrm{Effort}_2(t_2 + t_{1e}) - e_1$, and the total amount of effort applied to the repair by the first two repairpeople is expressed as $e_2 = \mathrm{Effort}_2(t_2 + t_{1e})$.

Finally, the amount of time the third repairperson would have to work to produce e_2 work is given by $t_{2e} = \mathrm{Effort}_3^{-1}(e_2)$. If t_0 is the time at which the third repairperson begins working, then the probability that she will finish in t time is given as

$$d(t_0 + t) = \frac{\mathrm{Delay}_3(t + t_{2e}) - \mathrm{Delay}_3(t_{2e})}{1 - \mathrm{Delay}_3(t_{2e})}.$$

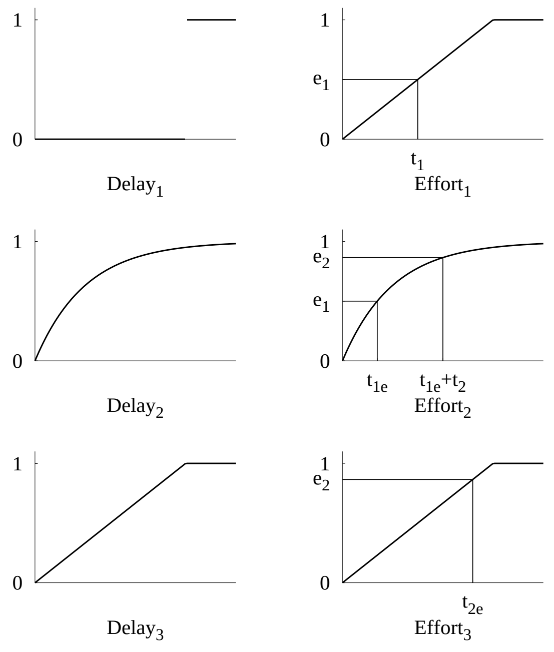

Figure 2. Example problem illustrated.

5 Model solution

The simulation of a model using the Möbius execution policy in its full generality is straightforward and has little additional performance overhead when compared to standard practices. The execution policy's greatest source of additional overhead comes from the possibility of having to perform an evaluation of the effort function and its inverse. The evaluation of the effort function must be performed only for disabling events when MTE or WE is preserved; otherwise, there is no additional overhead.

The evaluation of the inverse of the effort function occurs when an action becomes enabled and either WE or MTE is not zero and must be preserved. It is frequently the case (2 of our 3 repairpeople) that for continuous distributions, the delay distribution function is a good choice for the effort function. Note that performing the evaluation of the inverse of the delay distribution is commonly used to generate random numbers for many distributions, and can be done efficiently, so most simulators already provide many commonly used inverse functions. Again, the evaluation is little overhead, and is only incurred when the unique features of this execution policy are used.

As compared to implementing pri, when an action that is enabled becomes disabled and then becomes enabled, the pri method requires the multiplication and division of a real number, while our approach requires the evaluation of a function and its inverse. Our approach results in a slightly larger overhead than the pri approach. However, it allows for much greater flexibility, and the overhead is only necessary when the full flexibility of Möbius is needed. As we previously noted, no current formalism's execution policy takes advantage of the full generality of the Möbius execution policy. Formalisms may use "properties" (defined in [7]) to state the restrictions of the Möbius execution policy and avoid all additional overhead. Thus, Möbius provides a mechanism for avoiding any overhead penalty, except when the overhead is truly needed to express the intended behavior.

We have not addressed the issue of analytical solutions based on our approach. Again, through the use of properties, the analytic approach that others have explored for certain policies (e.g., [4, 9]) can be implemented within the Möbius framework. Since policies that are analytically tractable can be expressed in Möbius, and since the knowledge of the particular policy that an action implements can be preserved through properties, Möbius indirectly supports these analytic solutions.

6 Conclusion

The Möbius execution policy is able to capture the behavior of many different formalisms and execution policies, and is able to integrate them in a consistent way. In doing so, it is both a unification and significant generalization of existing policies. In particular, we are able to integrate many different execution policies, namely prd, prs, pri, and reactivation, into our execution policy in a consistent way. We are also able to integrate state-dependent behavior into our approach, in that every decision can be state-dependent. By giving the user complete control and flexibility, we provide a mechanism for resolving any ambiguities that arise in any state-dependent behavior. We relaxed the assumption that work proceeds at a constant rate and developed a mechanism for translating work between disparate delay distributions.

We described an interesting example (in Section 2.4) that showed the inability of previous work to capture realistic behavior, and illustrated how our approach solves the problem posed in this example. We built a simple, conceptual model for describing the limitations of other policies. We then formalized the discussion by quantifying the various behaviors with five action variables and a set of rules. We then enumerated all the possibilities and related them to other execution policies and examples.

In addition to solving the practical problem of integrating various execution policies, we were able to generalize them in many respects. This generalization gives us insight into behaviors and policies previously not considered. Finally, we described the little or no overhead involved in implementing this approach in a simulator, and explained how we are able to support the analytical solution of policies previously considered.

References

[1] M. Ajmone Marsan, G. Balbo, A. Bobbio, G. Chiola, G. Conte, and A. Cumani. The effect of execution policies on the semantics and analysis of stochastic Petri nets. *IEEE Transactions on Software Engineering*, 15:832–846, 1989.

[2] 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.

[3] A. Bobbio, V. Kulkarni, A. Puliafito, M. Telek, and K. Trivedi. Preemptive repeat identical transitions in Markov regenerative stochastic Petri nets. In *6th International Conference on Petri Nets and Performance Models (PNPM '95)*, pages 113–122, Durham, North Carolina, USA, Oct. 1995.

[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] C. G. Cassandras. *Discrete Event Systems: Modeling and Performance Analysis*. Aksen Associates Incorporated Publishers, Homewood, IL, USA, 1993.

[6] 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.

[7] D. D. Deavours and W. H. Sanders. Möbius: Framework and atomic models. In *Proceedings of Petri Nets and Performance Models (PNPM 2001)*, Aachen, Germany, Sept. 2001.

[8] 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.

[9] R. German. *Performance Analysis of Communication Systems with Non-Markovian Stochastic Petri Nets*. Wiley-Interscience Series in Systems and Optimization. John Wiley & Sons, Chichester, England, 2000.

[10] P. W. Glynn. A GSMP formalism for discrete event systems. *Proceedings of the IEEE*, 77(1):14–23, Jan. 1989.

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

[12] 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.

[13] 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.

[14] 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.