FORMAL SPECIFICATION OF THE MÖBIUS MODELING FRAMEWORK

BY

DANIEL DUANE DEAVOURS

B.S., University of Illinois, 1995
M.S., University of Illinois, 1997

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2001

Urbana, Illinois

FORMAL SPECIFICATION OF THE MÖBIUS MODELING FRAMEWORK

Daniel Duane Deavours, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 2001
William H. Sanders, Adviser

Modern computer and communication systems are becoming increasingly complex, and predicting the performance, dependability, and performability of such systems is challenging. A number of modeling formalisms have been developed to model certain types of systems, or aspects of a complex system, but no single formalism has proven to be ideal for all systems. Furthermore, no single way of solving models has been shown to be universally applicable and effective.

We have developed Möbius to address these challenges. Möbius is a complete, coherent, and consistent framework for building a software tool for the performance and dependability evaluation of complex, discrete-event systems. Möbius allows modelers to create models using a variety of formalisms supported by the framework. Models, possibly expressed in different formalisms, can then be composed together to form larger models. This allows users to use the formalism best suited for modeling a portion of the overall model. Furthermore, formalisms used to compose models together can operate on models without knowledge of the formalism that was used to create the submodels. Finally, solvers interact with Möbius models, as opposed to formalism models. Careful use of "properties" allows the application of efficient solvers to models.

A number of features make Möbius unique. First, we are unaware of any other approach that allows models to be created using very different formalisms and then to be composed structurally to form a single new model using composition formalisms. While the Möbius framework defines a universe of possible models, we avoid trying to create a "universal formalism" and instead try to create a general framework that supports a large variety of modeling formalisms.

In this thesis, we describe the Möbius framework and illustrate its usefulness by showing mappings from many formalisms into the framework. We also address some fundamental issues in the field of modeling of discrete-event systems, namely an efficient well-specified checker and a general, structured execution policy. Finally, we have developed a formalism to illustrate many of the features developed for Möbius but not found in any previously existing formalisms.

To my parents.

ACKNOWLEDGMENTS

This work has been enabled by a number of funding agencies. Specifically, it was funded by the DARPA Information Technology Office (under contract number DABT63-96-C-0069), the National Science Foundation's Next Generation Software Initiative directed by Frederica Darema (under grant number 9975019), and the University of Illinois Motorola Center for High-Availability System Validation, which exists under the umbrella of the Motorola Communications Center.

Many of the creative elements of this work rose out of discussions with many fellow colleagues. It would be difficult to list them all. Certainly the most influential is my adviser, Professor William H. Sanders. Many members of the Möbius team contributed through the process of developing the software tool. Möbius team members include Amy Christensen, Graham Clark, Tod Courtney, David Daly, Salem Derisavi, Jay Doyle, G. P. Kavanaugh, Doug Obal, John Sowder, Aaron Stillman, Alex Williamson, and Patrick Webster.

This thesis has also been a team effort. Dr. Sanders has read a number of drafts and given valuable feedback and discussion. Jenny Applequist helped immensely with the editing.

Finally, I would like to thank my committee. Ravishankar K. Iyer, Steven S. Lumetta, Benjamin W. Wah, and chair William H. Sanders provided guidance and were especially helpful in helping me explain Möbius to the nonmodeling specialist.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

GLOSSARY OF SYMBOLS

$([a, b)]$	Any of the intervals $[a, b]$, $[a, b)$, $(a, b]$, or (a, b).
2^s	Power set of s, i.e., the set of all subsets of s.
A	Set of actions in an atomic model.
$\mathcal{A}$	Annotations: extra information provided by the solvers about results.
$A_{C_{\min}}(\varsigma)$	The set of actions $\{a \in A : c_a \in C_{\min}(\varsigma)\}$, where c_a is the clock of action a in some augmented state. Sometimes written $A_{C_{\min}}$ if ς is clear from context.
$A^r_{C_{\min}}(\varsigma)$	The set of actions $\{a \in A : c_a \in C_{\min}, a.Rank(\sigma) = r\}$ for some state $\sigma \in \varsigma$. Sometimes written $A^r_{C_{\min}}$ if ς is clear from context.
Act	The action components of a model, including actions, action functions, action partition groups, and action properties.
AF	Action function, which is composed of *Enabled*, *Delay*, *Effort*, *Rank*, *Weight*, *Complete*, *Interrupt*, and *Policy*. See Section 3.3.2 for formal description.
AG	Action partition group. A partition of the actions that indicates which actions compete probabilistically for completion if they have the same completion time and rank.
$AG^r_{C_{\min}}(\varsigma)$	The set of action partition groups with actions in $A^r_{C_{\min}}(\varsigma)$. Formally, $AG^r_{C_{\min}}(\varsigma) = \{G \in AG : G \cap A^r_{C_{\min}}(\varsigma) \neq \emptyset\}$.
AM	Atomic model, which includes state variable components, action components, and model properties.
AP	Set of action properties.
AS	Action state, which is composed of Start, Delay, Effort, WE, and MTE. See Section 3.3.2 for a formal description.
$\mathcal{B}_{\mathrm{f}}(\mathbb{R})$	The set of all finite sets of nonoverlapping intervals over $\mathbb{R}$.
bool	The set $\{true, false\}$.

$\mathcal{C}$	Impulse reward.
$C(\varsigma)$	Set of action "clocks" in a particular augmented state ς. Sometimes written C if ς is clear from context.
$C_{\min}(\varsigma)$	Equal to $\min C$. Sometimes written $C_{\min}$ if ς is clear from context.
$Ch(\sigma)$	The set of nodes that are children of σ in a configuration graph.
Closed	A reward control function that is true if the utility time interval is closed, i.e., includes the current time point.
CM	A composed model, formed by structurally joining several reward models to form a single, larger, monolithic reward model.
Complete	An action function that describes how the state changes when an action completes.
CSL	Abbreviation for "continuous stochastic logic."
CTL	Abbreviation for "computational tree logic."
DDD	Execution policy for discarding worker effort, minimum task effort, and delay distribution and effort functions.
DDP	Execution policy for discarding worker effort and minimum task effort, and preserving delay distribution and effort functions.
Dist	A reward variable measure that indicates whether the reward variable should be solved in distribution.
DPD	Execution policy for discarding worker effort, preserving minimum task effort, and discarding delay distribution and effort functions.
DPP	Execution policy for discarding worker effort and preserving minimum task effort and delay distribution and effort functions.
Delay	An action function describing the random variable between enabling and completion.
Delay	An action state; a "sampled" value of *Delay*.
δ	The reward variable automata transition function.
e	An event that is the 4-tuple (σ, τ, a, s'), including the value of the state variables, the sojourn time, the action that completes, and the state resulting from the action completion.
E	The set of all events. See *event*.
E'	A homomorphism of E that is the set of all untimed events. See *untimed event*.

Effort	An action function describing the work performed by the action over time.
Effort	An action state; a "sampled" value of *Effort*.
$EN(\sigma)$	The set of enabled actions, i.e., $\{a \in A : a.Enabled(\sigma) = true\}$.
Enabled	An action function that determines whether the action is enabled.
EndEvent	A reward control function that evaluates to true at the end of the utility time interval.
EndTime	A reward control function that indicates the length of the utility time interval.
EndType	A reward control function that determines whether the length of the utility time interval ends in a fixed amount of time determined by *EndTime*, or if the utility time interval ends on an event determined by *EndEvent*.
EndUtility	A reward control function that evaluates to true when the utility time has been exhausted.
$F_d(t)$	Delay distribution of an action's uninterrupted time to completion, taking into account the worker effort and minimum task effort.
G	Some action partition group in AG.
G_a	The unique action partition group containing a.
I	An interval over the real line of the form $([a, b)]$, for $a < b$, or $[a, a]$.
$\bar{I}$	An indicator random variable.
I_e	An indicator function that is 1 if e is true, 0 if e is false.
IM	The set of allowable impulse rewards, which is some countable subset of $\mathbb{R}$.
Interrupt	An action function that determines when an action may be interrupted.
Intervals	A reward variable measure that is a set of interval sets of the type $\mathcal{B}_{\mathrm{f}}(\mathbb{R})$ indicating that the reward variable should be solved for the probability of being in each of the interval sets.
IS	Set of initial states of state variables of atomic models.
$\bar{J}$	An interval indicator random variable.
Markov	A reward control function that is true if all the reward functions that are functions of events are independent of the sojourn time.
Moments	A reward variable measure that is a set of integers that indicates which moments a reward variable should be solved for.
MP	The set of model properties.

MTE	An action state representing the minimum task effort.
$\mathbb{N}$	Set of natural numbers, i.e., $0, 1, 2, \ldots$.
$\bar{N}$	A counter random variable.
$NS(\hat{\sigma}^c_\bullet)$	The set of stable states of some model reachable from $\hat{\sigma}^c_\bullet$ in a stable step.
ν	Nil or null element.
$NU(\hat{\sigma}^c_\bullet)$	The set of unstable states reachable from state $\hat{\sigma}^c_\bullet$ in a stable step.
p_{ij}	$\Pr[\sigma_i \rightarrow \sigma_j]$, where $\rightarrow$ implies $\exists a \in EN(\sigma_i) : \sigma_j = a_i.Complete(\sigma_j)$.
Parms	Model parameters, made up of parameter names and values.
PDD	Execution policy for preserving worker effort and discarding minimum task effort and delay distribution and effort functions.
PDP	Execution policy for preserving worker effort, discarding minimum task effort, and preserving the delay distribution and effort functions.
Π	The set of all properties.
Policy	An action function that describes the particular execution policy of the action.
PPD	Execution policy for preserving worker effort and minimum task effort, and discarding the delay distribution and effort functions.
PPP	Execution policy for preserving worker effort, minimum task effort, and delay and distribution functions.
$\Pr[\sigma_i \rightsquigarrow \sigma_j]$	The probability of going from state σ_i to state σ_j by some sequence of events in zero time.
$\Pr[\sigma_i \overset{\sigma_d}{\rightsquigarrow} \sigma_j]$	The probability of going from state σ_i to state σ_j by some sequence of events that includes the intermediate state σ_d in zero time.
$\Pr[\sigma_i \overset{\bar{\sigma}_d}{\rightsquigarrow} \sigma_j]$	The probability of going from state σ_i to state σ_j by some sequence of events that does not include the intermediate state σ_d in zero time.
$\Pr[\sigma_i \rightsquigarrow \sigma_j, im]$	The probability of going from state σ_i to state σ_j and obtaining impulse reward im by some sequence of events in zero time.
P_S	Initial state probability distribution of state variables.
$\mathcal{R}$	Rate reward.
$\mathbb{R}$	Set of real numbers.
$R(\sigma_0)$	The set of reachable states in some model given initial state σ_0.
RA	Reward variable automata functions, including rt and δ.

Rank	An action function used to determine which action completes first if multiple actions are scheduled to complete at the same time.
RC	The reward control functions of a reward variable. Includes *RVType*, *UtilityType*, *UtilityTime*, *StartType*, *StartTime*, *StartEvent*, *Closed*, *EndType*, *EndTime*, *EndEvent*, *EndUtility*, and *Markov*.
Res	A mapping from a reward variable to the computed solution of various reward variable measures and annotations.
RF	Reward variable functions, including *RA*, *RC*, *RMeas*, and *RP*.
ρ	The state of the reward variable automata.
ρ_0	The initial state of the reward variable automata.
RM	A reward model, including an atomic model, R, RF, ρ_0, and RMP.
RMP	Reward model properties.
RS	The reward structure functions, including $\mathcal{C}$ and $\mathcal{R}$.
rt	The type of the reward variable automata; similar to state variable type.
RMeas	Reward variable measures, including the elements *Dist*, *Moments*, and *Intervals*.
RP	Reward variable properties.
RVType	Determines the type of the reward variable: an instant-of-time, interval-of-time, or time-averaged interval-of-time reward variable.
S	Set of state variables.
σ	The state of the state variables of a model.
$\hat{\sigma}$	A stable state, a state in which no zero-timed actions are enabled.
$\hat{\sigma}_\bullet^c$	A stable state σ in which the minimum completion time is c; used to label the root of a configuration graph.
SP	Underlying stochastic process of a Möbius model, defined as a sequence of events.
$SR(\sigma_0)$	The set of stable reachable states given initial state σ_0.
Start	An action state that describes the time of the last enabling or interrupting event.
StartEvent	A reward control function that evaluates to true at the start of a utility time interval.
StartTime	A reward control function that determines the instants or intervals of the reward variable utility time.

StartType	A reward control function that determines whether the beginning of the next instant or interval will occur at a fixed time or be determined by model events.
SV	The state variable components of a model.
SVP	Set of state variable properties.
t_i	The time of the ith event.
t_{je}	The larger of t_{jw} and t_{jm}, used to calculate MTE.
t_{jm}	The effective minimum task time of action a_j, used in an intermediate calculation. Sometimes written t_m if a_j is clear from context.
t_{jw}	The effective worker time of action a_j. Sometimes written t_w if a_j is clear from context.
T	Set of types that define a state variable. See Figure 3.1.
τ_i	The sojourn time of the ith event, i.e., the amount of time the model spends in state σ_i.
type	The type function, indicating the range of values the state variable may take on.
U	A set of nonoverlapping intervals over $\mathbb{R}$; $U \in \mathcal{B}_{\mathrm{f}}(\mathbb{R})$.
$UR(\sigma_0)$	The set of reachable unstable states of some model given initial state σ_0.
UtilityTime	A reward control function used for computing the length of a utility time interval.
UtilityType	A reward control function that determines whether the utility time is fixed and determined by *StartTime*, or variable and determined by a number of other reward control functions.
$\bar{V}_U$	A reward variable calculation for an instant-of-time reward variable defined over utility time U.
$\bar{W}_U$	A reward variable calculation for a time-averaged interval-of-time reward variable over utility time U.
WE	An action state representing the worker effort.
Weight	An action function used to probabilistically choose among actions that have the same completion time and highest rank, and belongs to the same action partition group.
X	The random variable describing the unconditional time to an action's completion.

$\bar{Y}_U$	A reward variable calculation for an interval-of-time reward variable over utility time U.
$\mathbb{Z}$	Set of integers.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Software environments for predicting the performance and dependability of complex computer systems and networks have become widespread, and their use has become integral to the design and development of such systems. The capabilities of these modeling tools have increased greatly over the last two decades, both from increases in computer speed and memory capacities, and from the development of more sophisticated and efficient modeling techniques. Despite these considerable advances, the scale and complexity of systems currently being designed have more than offset the advances in modeling. Modern systems are a complex combination of computer hardware, networks, operating systems, and application software. They include distributed computing systems, accounting for all system components, including the software application, the operating system, and the underlying computer and communications hardware.

Modeling formalisms have been developed to evaluate performance and dependability of smaller systems, or for single aspects of large and complex systems. For example, formalisms and solution techniques have been developed to address software performance, network protocol performance, and computer reliability. There are also many ways of solving models, and many formalisms are tailored so that efficient solution methods are always possible for models of those formalisms. Except for general-purpose formalisms, most formalisms are unable to express the behavior of many of the different subsystems that make up a modern, complex system, and except for "brute force" simulation, no solution method is capable of always solving any model. Even though simulation guarantees that a solution is possible, it may not always be practical. Evaluating the system as a whole becomes a special challenge that cannot be solved with a single modeling formalism or solution technique. This calls for the development of modeling frameworks and software environments that can address these

unique problems.

We believe that the best approach to addressing these problems is to develop a framework by which a number of different modeling formalisms and solution techniques can be tightly integrated into a single modeling tool or software environment.

Since different parts of a system may be represented in different formalisms, the ideal framework should provide a method by which multiple heterogeneous models (models built using different formalisms) can be composed together, with each component model representing some portion of the system. For example, one model may be of the software system, another of the hardware, and a third of the network and protocols. It may be best to represent each of these systems with a different formalism. Each of the models must interact with the others, so a composition technique should permit models of different formalisms to interact. The ways in which models may interact may include sharing of state, sharing of events, or sharing some results (some measurement taken on a model). These techniques should be scalable in the sense that the solution of the entire composed model should, if possible, take fewer computational resources to solve than an equivalent unstructured model.

Ideally, a framework should also support methods to combine models at different levels of detail. Furthermore, it should support multiple model solution methods, including both simulation and analysis, that are efficient and permit the solution of complete models of complex computing and communication systems, and the applications running on such systems. Finally, a framework should be extensible. This means that it should be possible to add new modeling formalisms, composition and connection methods, reward specification languages, and solution techniques to a software environment that implements the framework without significant changes to existing tool components.

1.2 Modern Performance/Dependability Tools

While an ideal modeling tool may seem beyond the capability of current research, dramatic progress has been made towards creating frameworks that have the desired capabilities. This suggests that it may be possible to develop a framework and software environment that meet many of our goals.

1.2.1 Single formalism, multiple solutions approach

In the historical progression of software tools, one of the major advances was the development of tools that support a single high-level formalism for specifying models that provides several solution methods for solving models expressed in the high-level language. The appro-

priate solution technique may be chosen depending on the characteristics of the particular model. While these tools do not support the multiple-model specification and composition methods needed in a complete framework, they do implement multiple solution methods, in recognition of the fact that no single solution method is sufficient for all models.

One kind of tools that are in this category is those that use some sort of queuing networks as their specification method. These tools include DyQN-Tool$^+$ [1], which uses dynamic queuing networks as its high-level formalism. Another tool, HIT [2], uses a homogeneous, structured paradigm for specifying systems. LQNS [3] uses layered queuing networks as its specification language. QNAP2 [4] allows users to specify a model as a network of service stations. Finally, RESQ and RESQME [5] use extended queueing networks as their specification method. In most cases, these tools support both simulation and product-form solution methods.

Another kind of tools in this category is those based on stochastic Petri nets and extensions. There are many tools in this category; see [6], [7] for a comprehensive list. They include DSPNexpress [8], GreatSPN [9], QPN-Tool and HiQPN-Tool [10], SPNL [11], SPNP [12], SURF-2 [13], TimeNET [14], and *UltraSAN* [15], among others. In each case, the tool supports model specification using some, possibly hierarchical, variant of stochastic Petri nets, and provides analytic/numerical solution of the underlying Markov process. Some of these tools also support simulation as a solution method.

Finally, this category includes a number of other tools that use other model specification approaches, sometimes tailored to particular application domain specifications. These tools include DEPEND [16], Figaro [17], HARP [18], HIMAP [19], and SAVE [20], which all focus on evaluating the dependability of fault-tolerant computing systems. They also include SPE·ED$^{\text{TM}}$ [21], which aims to aid in software performance engineering, and TANGRAM-II [22], which evaluates computer and communication systems using analytical/numerical methods.

For their intended applications, these tools are useful within the limitations of the solution methods that are implemented in them. However, not one of them individually meets the goals for an integrated performance and dependability modeling framework. Furthermore, extending any of these tools to implement such a framework would be difficult, since each of them was developed using a particular modeling formalism, and was intended for a single or small number of specific solution techniques, rather than with the aim of extensibility. New performance/dependability modeling frameworks and software environments are needed if we are to succeed in evaluating modern computer systems and networks. Two approaches have been taken to achieve this goal: incorporation of existing tools, and development of new tools. We discuss each of these below.

1.2.2 Multiple formalisms, multiple solutions approach

The first approach is to create a software environment that facilitates the combination of several of the tools of the type described above in a single environment. A perspective of this idea, in the context of software performance engineering, can be found in [23]. We are aware of three tools that take this approach. One tool, called IMSE (Integrated Modeling Support Environment) [24], is a support environment for performance modelers that contains tools for modeling, workload analysis, and system specification. Another tool, called IDEAS (Integrated Design Environment for Assessment of Computer Systems and Communication Networks) [25], [26], aims to give an analyst a broad range of modeling capabilities without requiring that the modeler learn multiple interface languages and output formats. The last tool, called Freud [27], has aims similar to those of IMSE and IDEAS, but focuses on providing a uniform interface to a variety of web-enabled tools.

In some of these tools, the primary focus is on providing both a common graphical interface for each tool, and a common interface for reporting results. In other words, the primary contribution towards the goals we outlined above is that these tools provide a method by which results from one tool may be used as inputs to another tool. While this approach is an important step towards building an integrated software environment that meets the requirements outlined above, they are inherently limited in the way models expressed in one tool can interact with models expressed in another tool, since the only way the two models can exchange information is via the output of an individual tool. Furthermore, the degree to which a user interface can be created depends on the similarity of the tools. In short, we believe that these tools take an important step towards solving the difficult problems described above, but that they are limited in how models may interact; furthermore, the degree to which different parts of a system may interact is substantially limited by the fact that the constituent tools were not designed to be integrated.

The second approach towards building an integrated performance and dependability modeling environment is to start from scratch and define a modeling framework that can accommodate multiple modeling formalisms, multiple ways for models of different formalisms to interact, and multiple solution methods. While this is a more difficult approach than building an environment out of existing tools, it has much more potential to integrate models expressed in different formalisms, while preserving and exploiting the special structural properties of a particular modeling formalism within the framework.

The earliest attempt to do this, to the best of our knowledge, was the combination of multiple modeling formalisms in the SHARPE modeling tool [28], [29]. In the SHARPE modeling framework, models can be expressed as combinatorial reliability models, directed

acyclic task precedence graphs, Markov and semi-Markov models, product-form queueing networks, and generalized stochastic Petri nets. Models expressed in SHARPE may share results in the form of exponential-polynomial distribution functions. This is an important step towards the development of an integrated performance/dependability modeling environment, in that it uses the exchange of results to obtain (overall) solutions to models expressed in multiple formalisms.

Another software environment that integrates multiple modeling formalisms in a single software environment is SMART [30], [31]. SMART supports the analysis of models expressed as stochastic Petri nets and queueing networks, and the tool is implemented in a way that permits the easy integration of new solution algorithms. The interface to the tool is entirely textual, and models expressed in possibly different formalisms can exchange information in the form of results, perhaps repeatedly, in the form of fixed-point iteration. Like SHARPE, SMART is an important advance in that it uses the exchange of results to permit the solution of models made up of multiple submodels expressed in different formalisms.

The DEDS (Discrete Event Dynamic System) toolbox [32], [33] also integrates multiple modeling formalisms into a single software environment, but it does so very differently from the two tools that we just described. The DEDS toolbox supports several formalisms including queueing networks, generalized stochastic Petri nets, and colored Petri nets. The DEDS toolbox converts models expressed in different modeling formalisms into a common "abstract Petri net notation" [32], [34]. Once expressed in this abstract formalism, models may be solved using a variety of functional and quantitative analysis approaches for Markovian models.

All of these projects show that it is possible to build a modeling framework and software environment that integrate models expressed in multiple formalisms more closely than would be possible if one integrated existing tools. In the case of SHARPE and SMART, composite models are built by exchanging results obtained through the solution of possibly heterogeneous submodels. In the case of the DEDS toolbox, the integration is obtained by converting the models into a common abstract notation. The DEDS approach has more potential than other approaches, because it allows models written in different formalisms may have much closer interaction, since they can interact as they execute. With DEDS, models can exchange messages, such as task arrivals, instead of merely exchanging solutions, such as throughputs.

The next step in increasing the generality of modeling tools is to build a modeling framework without presupposing what types of modeling formalisms would be supported, what methods would be used to combine submodels, or what solution methods may be used. If we could do this, we could build build a tool that would support the evaluation of complex computer and communication systems, and would also be extensible. While this goal is

probably impossible to achieve completely in practice (since certain assumptions must be made to implement a conceptual framework), it guides us in the development of the Möbius modeling framework.

1.2.3 The Möbius approach

The approach we take with Möbius [35] is an integrated multiformalism multisolution approach. Our goal is to build a tool in which each model formalism and solver is, to the greatest extent possible, modular, in order to maximize potential interaction. This goal is attainable because many operations on models, such as composition and connection (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 [36] has been used exclusively with stochastic activity networks (SANs). This exclusivity is artificial, and in the Möbius tool, Replicate/Join can be used with virtually any formalism that can produce a Markov chain, such as the stochastic process algebra PEPA [37].

Another advantage is the ease with which new approaches may be integrated into the tool. Perhaps the most convincing argument for this comes from our own experience. To the extent possible, we have incorporated new research results into our successful modeling tool, *UltraSAN*. For a number of practical reasons, many of our recent research results have only been developed to a prototype, and are not generally available to other users (for example, [38] – [42]). In particular, we would have liked to develop a new graph composition formalism of [42], but found it difficult to include in *UltraSAN* because of the inherently closed nature of the software design. Möbius has been designed to include these modeling techniques, as well as to be extensible to include future techniques.

Another one of our goals is to extend the state of the art in modeling expressiveness, convenience, and solution. When we say we wish to increase modeling expressiveness, we mean that we wish to increase our ability to express aspects of system behavior that were difficult or impossible with previously existing techniques. We are pursuing this goal through the development of new modeling formalisms. By "modeling convenience," we mean the level of ease with which a user can use a tool to create and solve models. This is the goal of the integrated software environments discussed above. We also wanted to create a vehicle for developing new model solution techniques and integrating them into a usable tool.

The ability to add new components simply would benefit researchers and users alike.

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

While we would like Möbius to have a great deal of flexibility, we would also like to 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.

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

1.3 The Möbius Framework

The Möbius framework is the formal specification of an environment in which multiple modeling formalisms and solvers can interact. As we stated earlier, the goal of the Möbius framework is to maximize the amount of formalism-formalism and formalism-solver interaction. For example, on-the-fly solution methods [43] have never been applied to PEPA [44] models, because there has never been an on-the-fly solver developed for PEPA. In Möbius, the PEPA formalism and the on-the-fly solver could both be integrated into the framework. The on-the-fly solver would then be able to solve PEPA models.

1.3.1 Framework description

The Möbius framework is an abstract machine specification together with an execution policy that precisely describes the stochastic behavior of a model. Models are expressed in particular formalisms within the Möbius framework. In order to be compatible with the

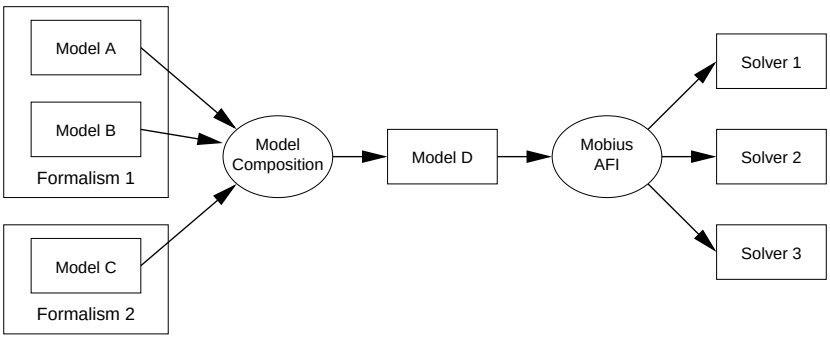

Figure 1.1: Models, formalisms, and framework.

framework, a modeling formalism must represent the various model components as framework components. In that way, models of different formalisms and solvers can interact, because they are interacting with framework components. Our goal is to develop a framework that is general enough that many useful formalisms may be expressed within it or translated directly to it.

Figure 1.1 illustrates the relationship between models, formalisms, and solvers within the framework. Users of the Möbius tool express models in the tool using a particular modeling formalism. The formalism editor, in turn, translates each component of the model into a corresponding component or components of the framework. The user is able to utilize all the advantages of a particular formalism, while at the same time having access to all the other components of the tool, such as various composition, measure, and connection methods, as well as a variety of solvers. This also allows the tool to be extensible, because we can add new formalisms and solvers with a minimum of impact on existing formalisms and solvers.

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

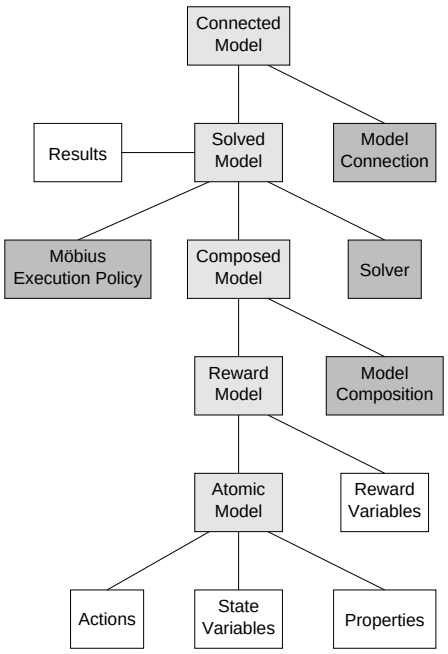

Figure 1.2: Möbius framework components.

1.3.2 Framework components

In order to define the framework, we must abstract away many of the concepts found in most formalisms. We also must generalize the process of building and categorizing models. We begin with an illustration, shown in Figure 1.2, that outlines the various components within the Möbius framework.

The first step in the model construction process is to generate a model or submodel using some formalism. This most basic model is called an *atomic model*, and is made up of state variables, actions, and properties. *State variables* (for example, places in the various stochastic extensions to Petri nets, or queues in queueing networks) hold state information about a model, while *actions* (such as transitions in stochastic Petri nets (SPNs) or servers in queueing networks) are the mechanism for changing model state. *Properties* provide information about a model that may be later used as a condition of the applicability of some solvers, or to make the solution process of some solvers more efficient. The atomic model is the topic of Chapter 3.

After a model or submodel is created, frequently the next step is to specify some measures of interest on the model using some reward specification formalism, e.g., [45]. The Möbius framework captures this pattern by having a separate model type, called *reward models*, that augments atomic models with reward variables. Earlier definitions of the framework called reward models *solvable models* [35], [46]. 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 of this include the Replicate/Join composition formalism [36] and the graph composition formalism of [42], in which symmetries may be detected and state lumping may be performed on the fly. Both of those formalisms have been implemented in the Möbius tool [47]. Other composed model techniques include synchronization on actions, which is found, for example, in stochastic process algebras (SPAs) such as PEPA [44], as well as in stochastic automata networks (also SANs, e.g., [48]) and superposed generalized stochastic Petri nets (superposed GSPNs, or SGSPNs) (e.g., [49], [50]). Note that the composition techniques do not depend on the particular formalism of the atomic models that are being composed, provided that any necessary requirements are met.

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

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

The computed solution of 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.

For each subsystem $S \in \mathcal{S}$
    Choose appropriate formalism F to represent S.
    Create model M_F^S, model of S using formalism F.
    Create appropriate reward variables on M_F^S.
While $|\mathcal{S}| > 1$
    Choose $S_i, \ldots, S_j \in \mathcal{S}$
    Compose $M_\bullet^{S_i}, \ldots, M_\bullet^{S_j}$ into composed model $M_\bullet^{S_k}$
    Let $\mathcal{S} = \mathcal{S} \cup \{S_k\} \setminus \{S_i, \ldots, S_j\}$
end while
Solve $M_\bullet^{\mathcal{S}}$

Figure 1.3: Algorithm for creating and solving a nonconnected model.

A *connected model* is a set of reward models in which input parameters to some of the models depend on the results of other models in the set. This is useful for modeling using decompositional approaches, such as that used in [51]. In those cases, the model of interest is a set of reward models with dependencies through results, and the overall model may be solved through a system of nonlinear equations (if a solution exists).

A typical algorithm for building and solving a composed model is described in Figure 1.3. The algorithm describes the process of building a model by first decomposing the model into subsystems, then choosing the appropriate formalism for expressing each subsystem, and then joining models using model composition. Notice that composed models may themselves be composed.

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

1.4 Contributions and Limitations

We are considering discrete-event systems, in which much of the behavior, especially the timed behavior, can be described using random variables and random processes, and the state changes at discrete points in time. Thus, a necessary condition for a model to be accepted by the Möbius framework is that the stochastic process generated by the model must be a stochastic process that changes only at discrete points in time. Examples of these

systems include a computer hardware reliability model, a high-level software performance model, and a communications satellite availability model.

1.4.1 Contributions

Our primary contribution is a complete, coherent, and consistent framework for building a software tool for performance and dependability evaluation of discrete-event systems. The unique thing about the Möbius framework is that it is designed to facilitate modeling using multiple atomic model formalisms, multiple composed model formalisms that can operate on models written using different atomic or composed model formalisms, multiple reward variable formalisms, multiple solvers, and multiple connection formalisms.

We provide a general framework in which a large number of formalisms that we surveyed are able to easily map into the Möbius framework. We address several difficult technical issues that arose from the development of Möbius, notably the creation of a general execution policy that significantly generalizes the execution policy of many formalisms, and provides a common basis through which models with different execution policies can be joined. We develop actions that implement this new execution policy, and a flexible set of types for state variables. We also generalize many features available in reward variable formalisms. We provide a proof that allows us to use an efficient algorithm for performing the well-specified check. Finally, we develope a new formalism with many new features and capabilities to illustrate many of the Möbius features that currently do not exist in any formalism.

1.4.2 Comparison of model classes

The class of models that Möbius is specifically designed to capture (but not limited to) includes those models in which the underlying stochastic process is generalized semi-Markov, semi-Markov, Markov, or those that afford specialized solution methods such as product-form and combinatorial solution.

We can formally define the set of models that Möbius accepts. Let $\mathcal{F}$ be the set of all modeling formalisms of discrete event systems, and let $\mathcal{M}_F$ be the set of all models that are expressible in the formalism $F \in \mathcal{F}$. We can say then that the universe of models $\mathcal{M}$ can be constructed as $\mathcal{M} = \cup_{F \in \mathcal{F}} \mathcal{M}_F$.

Now let $\widehat{\mathcal{M}}$ be the set of all models that Möbius accepts. Certainly, $\widehat{\mathcal{M}} \subset \mathcal{M}$. However, it is interesting to try to categorize and classify and classify which models are or are not in $\widehat{\mathcal{M}}$. Since models are expressed in formalisms and Möbius models must be expressed in the Möbius framework, there must be some "translation" process that converts a formalism

model to a Möbius framework model. Usually this translation is straightforward, and the translation is a process that is relatively simple to describe.

Formally, a sufficient condition for a model $m \in \mathcal{M}$ to be Möbius model is that there exists some mapping $f : m \mapsto \hat{m}$, where $\hat{m} \in \widehat{\mathcal{M}}$. The mapping f must be such that the measurable behavior of model $\hat{m}$ is identical to that of m

There are some formalisms F for which $\forall m \in \mathcal{M}_F, \exists f : \mathcal{M}_F \rightarrow \widehat{\mathcal{M}}$, that is, every model expressible by a formalism F may also be expressed in the Möbius framework. These include formalisms such as GSPNs or SANs. We also know that there are some formalisms for which some (but not all) models expressible in that formalism are not expressible in the Möbius framework; i.e., $\exists m \in \mathcal{M}_f$ s.t. $\nexists f : m \mapsto \hat{m} \in \widehat{\mathcal{M}}$. An example of this is an open queue with infinite servers implementing a complex preemption policy, because it would require an infinite number of Möbius actions, and Möbius models are limited to a finite number of actions.

An interesting class of models is called the *generalized semi-Markov process* (GSMP) [52], [53]. Is $\mathcal{M}_{\text{GSMP}} \subset \widehat{\mathcal{M}}$? This is an interesting question, and difficult to answer precisely because the definitions of GSMP given in [52], [53] are not as precise as we would like. If one takes the formal definition as presented, then we believe that Möbius supports GSMP models, and one can view Möbius as "GSMP with structure," that is, a GSMP-like framework with additional structure exposed to facilitate efficient solution for special GSMP subsets. The only technical difference between GSMP and Möbius is that a GSMP delay distribution can be selected as a function of events, while in the Möbius framework, the delay distribution is a function of the current state. This is not limiting, since the state of a Möbius model can always be augmented to include the last event, as is done with activity markings in SANs.

GSMPs have been developed as a formal description of various simulation languages. As such, some subtleties are sometimes lost or ignored so that simulation languages can be a superset of GSMPs. For example, decisions may be based on a global timer, or behavior may be altered based on clock values. These abilities are not strictly within GSMPs, but are available in several simulation languages for which GSMPs were developed as a method of description. Therefore, for any model developed in a simulation language that can be completely described as a GSMP and is made up of a finite number of elements, we believe there exists a mapping from that model to a Möbius model.

1.4.3 Scope and limitations

Our intention is to develop the Möbius modeling framework to meet all of the various goals stated in this chapter. There are a number of issues that may arise out of the creation

of such a framework that we have chosen not to address. For example, it may be possible to join two models of different formalisms in such a way that no known solution exists given current computational limitations. While the framework may allow this joining to take place, we do not address the question of whether a known solution exists. This issue is one that we leave to the developers of the composition formalisms. We have observed, however, that in many composition formalisms, the question of predetermining whether a reasonable solution exists is simply not addressed.

One limitation of the Möbius framework is that it requires formalism designers to find suitable mappings from formalisms to the Möbius framework, and that the mappings produce a Möbius model that can be solved efficiently. Ideally, the mapping should consist of an algorithm that is linear in the size of the original model, but there is no guarantee that an efficient algorithm always exists. Furthermore, even if an efficient algorithm exists, its complexity may be such that it is impractical to incorporate it in the tool. It may be that a framework may exist that is different from Möbius and that may allow simpler algorithms for the translation of models of various formalisms into that framework. However, we studied a large number of formalisms, and we believe that for most of them, an efficient and straightforward translation algorithm exists.

We do not provide a complete description of the mapping of a large number of formalisms into the Möbius framework. We do provide several partial examples and one full example using stochastic reward networks, but in general, we leave this problem to formalism developers. We believe that mappings exist and that algorithms to specify the mappings are not overly complex, however, implementing such a mapping for many formalisms is beyond the scope of this thesis. We rely on the existing implementation of several formalisms within the Möbius tool, and discussions about implementing others, as evidence that mappings exist and that a description of the mapping is not overly complex.

Furthermore, we do not address many issues involved in model connection formalisms. One such issue is the existence and uniqueness of a solution. While this is an important issue, there are a number of ways in which it can be addressed. These issues are best addressed by the various connection formalisms, and thus are outside the scope of the Möbius framework.

One of the limitations we discovered in developing a measure specification is that we were not able to find a commonality for all the measure specification languages that we surveyed. One class that remains outside of the Möbius framework is continuous stochastic logic. We address this problem in more detail in Section 4.2.

Another limitation of the Möbius framework is that it does not solve some of the issues that need to be solved in order for a tool to be implemented based on the framework. For example, the vocabulary of useful properties is left up to the formalism and solver designers.

This flexibility is necessary for the framework to be extensible, but one could argue that it shifts some of the difficult problems from the framework designer to the formalism and solver designers. However, we believe that the fact that a number of formalisms and solvers have been implemented within the Möbius tool indicates that this is not overly burdensome.

1.5 Overview

1.5.1 Outline

We begin in Chapter 2 by discussing one of the most basic issues in modeling: the execution policy. An *execution policy* is a formal set of rules for describing how a model evolves over time. We start with the most fundamental concepts and re-derive a range of possible and useful behaviors, including behaviors that, to the best of our knowledge, have never been used before in a general context. We introduce the key concept of using "effort," not time, to measure work. That allows us to handle a range of possible behaviors never before considered in a consistent, logical, and methodical way.

In Chapter 3, we begin to formalize the ideas of the Möbius framework and describe the "basic" model: the atomic model. Atomic models are made up of three Möbius model components: state variables, actions, and properties. State variables are the entities within a model that reflect the state or configuration of the system being modeled. State variables may have structure, so they may represent complex component configurations in a way that closely resembles the component being modeled. Actions are the entities that change the values of the state variables over time. Actions model how things change, and the way that actions interact with the rest of the model follows from our discussion in Chapter 2. Properties retain any special structural information that might otherwise be lost when the model formalism components are translated into Möbius framework components.

Chapter 4 describes the Möbius reward model. A reward model allows modelers to measure some aspect of the model. This is done using reward variables, which are able to measure the time a model is in a set of states, transitions between states (events), and sequences of events. For example, reward variables can measure model characteristics such as reliability, average response time, or expected profit. The reward model is a model that may be "solved" by solvers.

Once we have defined the reward model and described the execution policy, we can formally define the Möbius execution policy. This allows us to formally formulate the stochastic process of a Möbius model. This is the topic of Chapter 5.

In providing a formal specification for the Möbius framework, we faced some fundamental

problems that are broadly regarded as open problems within the field. One of them is the so-called "well-specified" problem. It addresses the issue of what to do when multiple events are set to occur at the same time and no ordering information is provided. This is especially important within the Möbius framework, because it is possible that models described in different formalisms may have events that are intended to occur at the same time. This is a new problem that arose from the new abilities afforded by Möbius.

We review several options, including the one we chose for the Möbius framework. This option is to provide a detection algorithm that can determine when the ordering determination is truly necessary. We know of only one previously existing algorithm for performing the well-specified check, and it is exponential in complexity [54]. We have discovered an algorithm that is linear in time complexity [40]. In Chapter 6, we will describe how we extended this work to include general timed actions and incorporate it into the Möbius framework.

Chapter 7 discusses several other model types described briefly in Section 1.3. These include composed models, solved models, parameterized models, connected models, and study models. To develop these model types, we define more model components, namely parameters and results. Parameters make it possible to change models simply by changing the value of one parameter, e.g., an arrival rate. Results are the computed solutions of reward variables. Connected models make use of results and parameters by using results from one model to compute the parameter values of other models.

Next, in Chapter 8, we describe a novel formalism that is designed to show that the Möbius framework has capabilities that do not exist in any other known formalism or collection of formalisms. Although designed for illustration purposes, it is a very powerful formalism that integrates many of the advanced constructs found in object-oriented programming languages and simulation languages, but also retains the ability to use analytical Markov-based solution methods if certain conditions are met. This new formalism, called DEF, is a unique compromise between simulation languages and stochastic Petri nets.

We conclude in Chapter 9, in which we summarize what we have accomplished, and speculate about future work that may enhance the Möbius framework.

CHAPTER 2

A FLEXIBLE EXECUTION POLICY

2.1 Introduction

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) [55], as well as reactivation [56], 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 multiformalism, multisolution nature of the Möbius framework makes it necessary to develop an execution policy that is flexible, and that 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 behaviors, 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, Möbius generalizes the various preemption policies and allows them to be completely state-dependent. For example, 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 binary independent decisions, resulting in eight possibilities. These possibilities capture and extend the various execution policy behaviors in a state-dependent way.

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 and allows for arbitrary changes in delay distribution, and does both of these things consistently. 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 chapter, we begin in Section 2.2 by reviewing previous work concerning both Möbius and other execution policies, and include a motivating example. Section 2.3 then shows the details of our solution for Möbius in the action state and execution policy rules. In Section 2.4, we derive the equations needed to describe the behavior of an action, and we discuss issues with solvers in Section 2.5. We conclude our discussion of the execution policy in Section 2.6.

2.2 Motivation

2.2.1 Möbius framework description

A detailed overview of the Möbius framework was provided in Section 1.3, but for this chapter, it is sufficient to know that Möbius makes it possible to use multiple formalisms to describe a single model. This means, for example, that one portion of a model may be described using one formalism, such as generalized stochastic Petri nets (GSPNs) [57], while another part of the model is described with a different formalism, such as PEPA [44]. Sub-models of different formalisms may be combined into a single, larger model using a technique called model composition, e.g., Replicate/Join [36], or SGSPNs.

2.2.2 Execution policies

One of the major challenges in developing the Möbius execution policy was to develop an execution policy that encompasses all the policies used in most 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 [55], [58]. 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 [59], 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 [56] include the concept of reactivation. An activitiy may reactivate as a function of both the state of the model, as long as the activity is enabled, and the state in which the activity last activated (became enabled or completed and remained 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 [52], [53] 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 [55] use an age and resample variable, we have additional variables, and with these variables we are able to address complex and useful behaviors systematically.

2.2.3 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 [55]. As defined in [55], 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., [60], 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 [56], 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 in 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 [46] that are able to capture these restrictions,

so a generalized execution policy does not preclude efficient solution. Furthermore, our immediate goal is to allow for the *expression* of interesting behaviors, not just behaviors for which we already know how to compute analytical solutions.

We propose such a generalized execution policy. One thing that such a policy requires is the relaxation of what we call the "constant work assumption." For an example that shows why such a relaxation is necessary, consider the formulation of GSMP given in [52], [53]. 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 that 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.

2.2.4 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 will 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 enabling and completion of the action, called Delay, is defined so that $\text{Delay}_1(t) = 0$ 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, $\text{Delay}_2(t) = 1 - e^{-\lambda t}$. 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 effective "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, $\text{Delay}_3(t) = t/b$ for $0 \leq t \leq b$. 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 [60], 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.

2.3 Realization of Möbius Execution Policy

2.3.1 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 2.2.4 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 complex behavior, we must develop mechanisms to describe it precisely. To do this, we define the notion of an "event."

2.3.2 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. However, since model composition can alter the behavior of the model, the next state may not be uniquely defined at the time an atomic model is constructed. Therefore, we include the next state as an element in an event.

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 $(\sigma, \tau, a, \sigma')$, $a_0 = a$.

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 en-

countered 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. (Note that a completing event is not a defining 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.

2.3.3 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 Section 3.3, and action state, which captures the current state of an action. We summarize them here for convenience. The action state is made up of the following five action state variables.

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

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.

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

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

Effort : $\mathbb{R} \rightarrow [0,1]$ The effort function is defined such that 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: Effort$(0-) = 0$ (to enforce causality), 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.

WE : $[0, 1]$ The worker effort is a measure of the amount of work the worker has performed on the task, and is computed through an evaluation of 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 WE = Effort$(t_1) = 0.5$.

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 WE $\leq$ 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.

2.3.4 Rules for action state

The rules regarding the start time are simple. If an action is enabled, the start time is the time of the most recent enabling, interrupting, or completing event. If an action is disabled, the start time is zero.

At any defining event, the delay function, effort function, 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 Delay$'$ = Delay and Effort$'$ = Effort. If they are both discarded, then Delay$'$ = $\emptyset$ and Effort$'$ = $\emptyset$. By using separate action state variables for effort and delay, we can 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 depends.

If the worker effort is preserved and the defining event is an enabling event, then WE$'$ = 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 = Effort^{-1}(WE). Let τ be the current time minus Start. Then WE$'$ = 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 WE$'$ = 0. Our repair example in Section 2.2.4 is an example of a situation in which we would want to preserve the worker effort. We also need to preserve worker effort for the prs policy.

If the minimum task effort is preserved, then MTE$'$ is set to the larger of MTE and what WE$'$ would be if the worker effort were preserved. If the minimum task effort is discarded, then MTE is set to WE$'$. This preserves the invariant WE $\geq$ MTE.

2.3.5 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 queuing 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 usefulness of PPD is illustrated in our example in Section 2.2.4.

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 [59]. 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.

2.3.6 Justification for variables

We 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 [53], [52] 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 [55] 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 variable 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

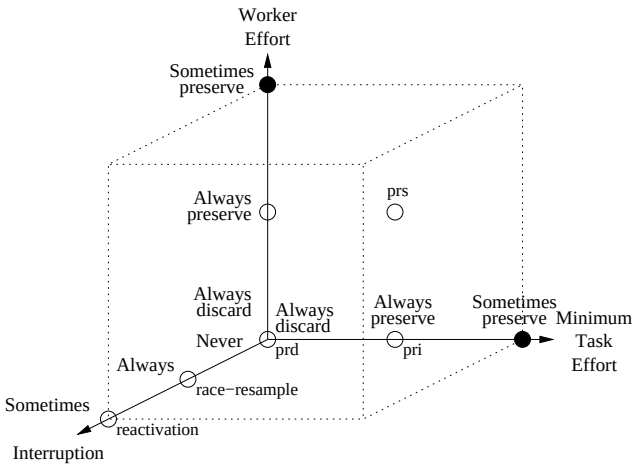

Figure 2.1: Möbius policy space.

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).

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 2.1. 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.

2.4 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.

2.4.1 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 nondecreasing. We need a unique inverse to compute the delay characteristics of an action.

For some time t, let $T_e = \{t : e = \text{Effort}(t)\}$ be the set of elements that satisfy the inverse. We choose the unique inverse to be the smallest time in T_e, that is, $t = \text{Effort}^{-1}(e) = \min T_e$. Finding a unique inverse is only a problem when the effort function is constant 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 constant interval, then we interpret it 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 is illogical.

2.4.2 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 WE $\leq$ MTE, $e_w \leq e_m$. Let $t_w = \text{Effort}^{-1}(e_w)$ and $t_m = \text{Effort}^{-1}(e_m)$. We know that because Effort is nondecreasing, $t_w \leq t_m$.

Let X be a random variable with the distribution function Delay. Let the action have a worker effort e_w and a minimum task effort e_m, and let $t_0 = \text{Start}$ 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 $t_w = \text{Effort}^{-1}(e_w)$, and t_m be the *effective minimum task time*, again computed as $t_m = \text{Effort}^{-1}(e_m)$. The effective delay distribution, $F_d(t_0 + t)$, gives the

probability that the action will complete by time $t_0 + t$.

We have the information 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 that the action will complete. We can write this precisely:

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

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 effective 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.

2.4.3 Solution to example

Using this solution, we are now able to solve the problem we posed in Section 2.2.4. 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 may be indicated through a state variable.

Let $\text{Effort}_1(t) = t/c$ for the first repairperson, $\text{Effort}_2(t) = 1 - e^{-\lambda t}$ for the second repairperson, and $\text{Effort}_3(t) = t/b$ for the third. The amount of effort performed by the first repairperson is calculated by $e_1 = \text{Effort}_1(t_1)$, 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 $t_{2e} = \text{Effort}_2^{-1}(e_1)$. If the second repairperson works for t_2 time, then the amount of work performed by the second repairperson is $\text{Effort}_2(t_2 + t_{2e}) - e_1$, and the total amount of effort applied to the repair by the first two repairpeople is expressed as $e_2 = \text{Effort}_2(t_2 + t_{2e})$.

Finally, the amount of time the third repairperson would have to work to produce e_2 work is given by $t_{3e} = \text{Effort}_3^{-1}(e_2)$. 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

$$F_d(t_0 + t) = \frac{\text{Delay}_3(t + t_{3e}) - \text{Delay}_3(t_{3e})}{1 - \text{Delay}_3(t_{3e})}.$$

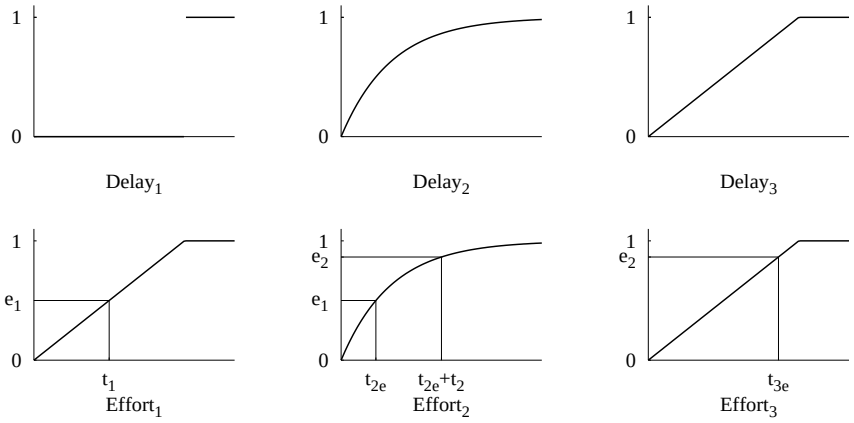

Figure 2.2: Example problem illustrated.

Figure 2.2 illustrates this example graphically.

2.5 Model Solution

The simulation of a model using the Möbius execution policy in its full generality is straightforward and has little more performance overhead than 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 (as is the case for 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 adds little overhead, which is only incurred when the unique features of this execution policy are used.

When an action that is enabled becomes disabled and then becomes enabled, the pri method requires the multiplication and division of a real number; 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" (see Chapter 3) 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., [55], [60]) 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.

2.6 Conclusion

The Möbius execution policy can capture the behavior of many different formalisms and execution policies and can integrate them consistently. In doing so, it is both a unification and significant generalization of existing policies. In particular, we can integrate many different execution policies, namely prd, prs, pri, and reactivation, into our execution policy consistently. We can also 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 2.2.4) 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 certain previously considered policies.

CHAPTER 3

ATOMIC MODELS

3.1 Introduction

Most modeling formalisms are ways of describing and mimicking system behavior. This is what one usually thinks of as a "model." For example, GSPNs, fault trees, and queueing networks are ways of describing systems. These formalisms create what we call "atomic models." Other formalisms exist that specify measures on models, structurally combine several models together, or combine models by sharing results. Those model types are discussed in later chapters.

Atomic models have several traits. First, an atomic model is created by a single modeling formalism. Atomic models mimic an entire system, or part of a system. The models can express the various system states and the various ways a model can change state over time. They usually do so by decomposing the system state into smaller "chunks" by representing the state of various simpler components. Formalisms do this in various ways, such as with places in stochastic Petri nets, or queues in queueing networks. Furthermore, the complex ways that a system can change state is usually decomposed into smaller processes, such as transitions in stochastic Petri nets, or servers in queueing networks.

Möbius captures these "chunks" of system state using state variables, and processes that change state using actions. State variables are discussed in Section 3.2, while actions and other associated components are described in Section 3.3. Section 3.4 then formally defines the atomic model. We illustrate the ability of Möbius to express SRN models in Section 3.5, and conclude in Section 3.6.

A framework is a substantial undertaking, and for any particular choice we made in defining what is within the framework, and, equally, what is outside of the framework, it is legitimate to question our choice. We do not claim that our choice is in some sense "complete," nor do we argue that we have always made the "right" choices by proof. We

simply offer evidence (often in the form of examples) that we studied many formalisms, and that most of them seem to fit nicely within this framework. We believe that this is the best that can be done, given that our measures of success include modeling convenience and the ability to facilitate efficient and appropriate interactions between model components. Also, due to space limitations, we cannot reference all of the relevant work, so we have chosen to present those references with which we are most familiar.

3.2 State Variables

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

Different formalisms represent state variables differently. For example, SPNs and extensions have places that contain tokens, so the set of values that a place can take on is the set of natural numbers. Colored GSPNs (CGSPNs) [61] have been extended so that tokens can take on a number of different colors at a place, making the value of a colored place a bag or multi-set. Queueing networks with different customer classes can have more complicated notions of state, such as those found in extended queueing networks [62], in which each job (customer) may have an associated job variable, which is typically implemented as an array of real numbers.

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

To do that, we had to create a framework in which we could create general types of state variables. By using a framework, we enjoy all the benefits of a framework that we discussed in Section 1.3. Specifically, solvers can interact with Möbius state variables, instead of with a variety of different formalism state variables. Composed model formalisms also need to interact only with Möbius state variables or actions. Finally, any efficiencies that may be gained through any structural knowledge can be preserved through the use of properties.

3.2.1 Components of state variables

In the Möbius framework, a state variable is made up of three components: a value, a type, and an initial value distribution, which includes a set of possible initial states. The value of a component represents a particular configuration that the modeled component may

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

Figure 3.1: Construction of the type set T.

be in, while the type of a state variable represents the range of values that the variable may take. The initial value distribution probabilistically gives the distribution of values at time 0. A distribution of values allows us to consider models whose initial states may vary. For example, it allows us to consider a system that is in steady state and measure the expected time until some important event.

Types

We begin by describing state variable types. Let $S = \{s_i\}$ be a set of state variables. (One can think of this as a set of state variable names.) Let T be the set of types that a state variable may take on, and let $type : S \to T$ be the type function. We construct T as the smallest set satisfying the rules shown in Figure 3.1. Here, $\mathbb{Z}$ is the set of integers, $\mathbb{R}$ is the set of reals, 2^t is the power set of t, and ν is a `nil` element. The meaning of $([a, b)] \in T$ is $[a, b], (a, b], [a, b), (a, b) \in T$.

We briefly describe some implications of the rules constructing T. Possible types are integers, a subset of integers, reals, an interval over the reals, pointers to state variables (including `nil`), a set of a type, a finite and infinite tuple of a type, 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.

Values

Naturally, state variables take on values, and the values of state variables change over time. As described informally earlier, the values that a state variable may take on are

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

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

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, or the initial state may be unknown. To capture this behavior, we include a set of possible initial states and a probability function describing the probability of being in each initial state. Let $IS = \{val_i\}$ be a finite set of initial values for the state variables. Let $P_S : IS \rightarrow [0, 1]$ be the initial state probability distribution so that $P_S(v) = \Pr[\text{Initial value of state variables is } v]$. This allows the initial state to be probabilistically distributed among different values.

3.2.2 Properties

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

Properties are nothing more than a set of symbols that have proprietary meaning. We write properties as <property>. Let Π be the set of all properties. The *state variable properties* is a function $SVP : S \rightarrow 2^{\Pi}$. An example property is <member P-invariant 1> $\in SVP(s_i)$, which may indicate that the state variable is part of a particular P-invariant; a state-space generator may take advantage of this property 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 the state variable is replicated.

Note that a property does not provide additional information about a model. Rather,

it provides information about a condition that is true, but may be difficult or expensive to determine. For example, one could test each action of a model to determine whether it is exponentially distributed in all states, but that may be expensive compared to the effort required if a property already states that this is true.

3.2.3 Definition

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

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

3.2.4 Examples

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

A GSPN [57] place has type $\mathbb{N}$, which is a subset of $\mathbb{Z}$, so the type of a GSPN place can be formed using rule 2.

The value of a CGSPN [61] 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. The number of tokens of a color in s is $\mathbb{N} \in T$ as shown above. Thus, the state of a colored place can be expressed as $C \times \mathbb{N} \in T$ by rule 7, where the first term represents the color, and the second term represents the cardinality.

SPAs are significantly different from SPN models because state variables are not as explicitly expressed in SPAs. For example, a PEPA [44] model state comprises the states of all the sequential components. Each sequential component may be represented as a state variable within the Möbius framework. Let some sequential component s have K stages. Each stage can be numbered $1, \ldots, K$. Note that $\{1, \ldots, K\} \in T$ by rule 2. We also note that discovering all the sequential components of a PEPA model involves some work, but this work is normally required for analysis, so it is a reasonable requirement.

Consider a finite FCFS queue with customer classes, where service time differs depending on the class [63]. Let C be the set of integers that enumerate the set of customer classes. We know $t_1 = \{0\} \cup C \in T$. (Value 0 is used to denote "no customer.") Then let t_2 be the k-tuple $t_1 \times t_1 \times \cdots \times t_1$. $t_2 \in T$ by rule 7. If the queue is an infinite queue, then $t_3 = t_1 \times t_1 \times \cdots \in T$ by rule 8.

Next consider a finite priority queue with a preemptive resume policy. An action (discussed in Section 3.3) stores the age information for the active job of each priority. The state variable representing the queue can be the same one we constructed for the CGSPN place.

Finally, we illustrate the extended queueing network (EQN) [62] job variable. A job

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

3.3 Actions

An action represents the basic unit of a model that facilitates the changing of the state of state variables. An action corresponds to a transition in SPNs [64], GSPNs [57], and other extensions, an action of an SPA (e.g., [44]), an activity of a SAN [56], and a server of a queueing network (e.g., [63]), for example.

Möbius actions are similar to state variables in that their goal is to provide an abstraction of the various concepts of actions present in most formalisms. State-change mechanisms of atomic model formalisms in the Möbius framework may be implemented using a subset of the functionality provided by actions. Note that it is the restriction of the possible generality that often allows for efficiencies in solution methods. For example, restricting the delay times to be zero or exponential is useful, because the underlying stochastic process is then Markovian. If behavior of a queueing formalism is limited 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 [55], for a set of rules to unambiguously define the underlying stochastic process that describes the behavior of a model in the Möbius framework. The execution policy is described in detail in Chapters 2 and 5, but we review it for convenience below.

Finally, like state variables, the Möbius action provides a common interface by which other model components (possibly of different formalisms) and solvers may interact. This allows for composition by synchronization, as is found in SPAs, stochastic automata networks (e.g., [48]), and superposed GSPNs (e.g., [49], [50]). These types of model composition

methods are possible within the Möbius framework.

3.3.1 Execution policy review

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

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

This execution policy allows us to implement all the various preemption policies including prs, prd, and pri, all of which are described in [55]; the concept of reactivation found in SANs [56]; and other execution policies not generally considered by others (see [65] for examples of these). Möbius actions must be defined so that they can utilize this general execution policy. We provide the action definition in the following section and then give examples of how state-change mechanisms of particular formalisms can be expressed as actions.

3.3.2 Action components

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

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

Figure 3.2: Action functions.

Action functions.

Formally, an *action function* is the mapping

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

where each of the terms in the codomain is a function, whose type is given in Figure 3.2. We use the symbol Σ to denote the set of state variable values, and E to be the set of possible events. An *event* takes the form of the 4-tuple $(\sigma, \tau, a, \sigma')$: a state, the sojourn time, the action that completes in σ, and the state resulting from the completion of a in σ. Note that since we frequently talk about a single action function as opposed to the set of functions, we adopt an object-oriented style of notation, with which we write $a.\mathit{Enabled}$ to mean the *Enabled* function of $AF(a)$.

Below, we briefly describe each of the action functions.

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

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

Effort : $\Sigma \to (\mathbb{R} \to [0,1])$ A function describing how work proceeds over time. This is discussed in greater detail in [65].

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

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

Figure 3.3: Action state.

Weight : $\Sigma \rightarrow \mathbb{R}^{\geq}$ A function used to select probabilistically among actions that are scheduled to complete at the same time, have the same rank, and belong to the same "action partition group" (see Section 3.3.3). The details are explained below.

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

Interrupt : $\Sigma \rightarrow$ *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 [65].

Policy : $\Sigma \rightarrow \{\text{DDD}, \ldots, \text{PPP}\}$ 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, preserve or discard minimum task effort, and preserve or discard the delay and effort functions. These policies are described in detail in [65].

Action state

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

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

where each element in the co-domain is a function given in Figure 3.3.

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

We describe each action state component below.

Start : $\mathbb{R}^{\geq}$ The time of the last enabling or interrupting event.

Delay : $(\mathbb{R} \rightarrow [0,1]) \cup \{\emptyset\}$ A value from the distribution function describing the (random)

uninterrupted time between enabling and completion of an action. This is the "sampled" value of $a.Delay$.

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

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

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

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

Initial action state

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

3.3.3 Action partition groups

Of all the enabled actions, the action that will complete first 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. (This is described formally in Chapter 5.) 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, and is the topic of Chapter 6.

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.

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 weights 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.

Let the *action partition groups*,

$$AG : 2^{2^A},$$

be a partition of the actions so that $\cup_{G \in AG} G = A$ and $\cap_{G \in AG} G = \emptyset$. Let $C = \{c_a\}$ be the set of clocks of the actions in some state σ that define the remaining times until enabled actions complete (see Section 5.2.3) and $C_{\min} = \min C$. The set of actions with completion time $C_{\min}$ is indicated by $A_{C_{\min}} = \{a \in A : c_a \in C_{\min}\}$. Let the action with the highest rank have rank r, i.e., $r = \max_{a \in A_{C_{\min}}} \{a.Rank(\sigma)\}$. Then let $A^r_{C_{\min}} = \{a \in A : c_a \in C_{\min}, a.Rank(\sigma) = r\}$, the set of actions with the smallest completion time and highest rank.

If $|A^r_{C_{\min}}| > 1$, then actions must compete probabilistically to determine which action completes first. However, actions only compete within the same action partition group. Let G_a be the set $G \in AG$ such that $a \in G$, the unique partition group containing a. If $A^r_{C_{\min}} \cap G_a = A^r_{C_{\min}}$, that is, if the actions in $A^r_{C_{\min}}$ all belong to the same partition group, then the model is exhibiting well-specified behavior (see Chapter 6), and the probability that a certain action a will complete first is determined by the expression

$$\Pr[a] = \frac{a.Weight(\sigma)}{\sum_{a' \in G_a} a'.Weight}.$$

Otherwise, if $A^r_{C_{\min}} \cap G_a \neq A^r_{C_{\min}}$, that is, if actions in $A^r_{C_{\min}}$ belong to different partition groups, then the well-specified check must be performed; this is explained in Chapter 6. If the result of the well-specified check is a "pass," then any action partition group may be chosen to complete first.

3.3.4 Properties

Let

$$AP : A \rightarrow 2^{\Pi}$$

be the *action properties* of a Möbius model. An example would 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 preemptive repeat identical execution policy. This information can be used to make simulation or analytic solution more efficient.

3.3.5 Definition of actions

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

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

3.4 Atomic Model Definition

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

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

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

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

3.4.1 On properties

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

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

Properties are the one area of Möbius in which components are not completely modular. A new formalism and solver may introduce new properties that are not understood by other composition, reward variable, and connection formalisms. In that case, each of the formalisms should be updated so it knows whether to preserve the new property. The rule that states that formalisms may not preserve properties that they do not understand keeps

Möbius safe. However, until all model components are updated to respond appropriately to the property, users may not have access to the full modeling power of Möbius. We believe that the cost of updating each tool component to respond to new properties is minor compared to the effort required to implement a new formalism or solver.

3.5 Example: SRN

We illustrate the use of Möbius in capturing an SRN [66] atomic model. Let **A** be an SRN model. **A** is defined in the usual way:

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

where $P = \{p_i\}$ is a finite set of places, $T = \{t_i\}$ is a finite set of transitions, D^+, D^-, and D° are the marking-dependent multiplicities of the input, output, and inhibitor arcs, g is the guard function, $>$ is the priority relation, μ_0 is the initial marking, λ is the rate function of the transitions, w is the weight function used for immediate transitions, and M is a set of measures. SRN measures are covered by the Möbius reward model discussed in Chapter 4.

SRNs have places which hold a nonnegative number of tokens. The number of tokens in each of the places pepresents the state of the model. Places correspond to Möbius state variables in the following way. Let $S = \{s_i\}$ be the state variables corresponding to places $\{p_i\}$. Let $\forall s_i \in S$, $type(s_i) = \{0, \ldots, \infty\}$ by Rule 2. The initial marking of an SRN model is given by μ_0. The initial state of the corresponding Möbius model can be defined so that $IS = \{val_{IS}\} : val_{IS}(s_i) = \mu_0(p_i)$ with $P_S(val_{IS}) = 1$. In SRNs, the function $\#(p_i, \mu)$ yields the number of tokens of place p_i in marking μ. This corresponds to the Möbius function $val_\sigma(s_i)$, the value of state variable σ_i in state σ. This allows us to use *val* and # interchangeably.

In SRNs, transitions change the number of tokens in places when the transition fires according to a number of rules. A transition t_i is said to be *arc-enabled* in marking μ iff the following holds.

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

The transition t_i is *enabled* iff

$$t_i \text{ is } \textit{arc-enabled} \wedge g_{t_i}(\mu) \wedge$$

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

where $>$ is the priority relation. The corresponding Möbius action can construct the function *Enabled* such that it is true iff the corresponding SRN transition is enabled.

The SRN transition has an exponential delay, which can also be zero-timed if $\lambda(\mu)$, the rate function of the transition, is infinity. Thus, $a_i.Delay(\sigma) = 1 - e^{\lambda(\mu)t}$ if $\lambda(\mu) < \infty$; otherwise, it is the step function.

When a transition t_i fires, the marking of the SRN changes in the following way.

$$\forall p \in P, \#(p, \mu') = \#(p, \mu) - D^{-}_{p,t_i}(\mu) + D^{+}_{p,t_i}(\mu)$$

The function $a_i.Complete$ can be constructed to implement the above expression.

Since the SRN transition implements only the prd policy, the effort function is irrelevant and $a_i.Policy = \text{DDD}$.

3.6 Conclusion

In this chapter, we described atomic models, which include state variables, actions, and various properties. We described state variables in detail, including types and values, and the set of initial states with their accompanying probability distribution.

To describe actions, we reviewed briefly the execution policy, which we describe in depth in Chapter 2, and then described each of the components of actions. They include the action (names), action functions, the action partition group, and action properties. We described each of the action functions, the action state, the action partition group, and action properties. The action state is not technically a component of actions, but is used in describing the Möbius augmented stochastic process (see Chapter 5). Action partition groups are used when multiple actions have the same completion time and share the highest rank, to determine probabilistic selection. Action properties may be used by solvers to aid in efficient solution.

Finally, we illustrated the use of the Möbius atomic model by describing a direct translation of stochastic reward nets into Möbius.

CHAPTER 4

MÖBIUS REWARD VARIABLES

4.1 Introduction

People build models of systems because they are interested in learning something about the systems. Möbius is specifically designed to aid in the prediction of performance, dependability, and performability. These various measures are expressed as measures of the timed behavior of the generated stochastic process of the model.

Let SP represent the underlying stochastic process of a Möbius model. We know that SP takes on a countable number of states, and that it is a continuous-time stochastic process. Now we introduce some measure of interest Y, which "sits on top" of SP, and monitors SP. The behavior of SP is sufficient to specify the output of Y. Syntactically, SP is a random process, i.e., it commonly takes the form $SP : \Omega \times \mathbb{R} \rightarrow \mathbb{R}$. Actually, we define two stochastic processes for two different purposes in Chapter 5. For measurement, we define a *measurable stochastic process*, and for defining the execution policy, we define and use an *augmented stochastic process*. For our discussion reward variables in chapter, we are only concerned with the measurable stochastic process, which we simply call *stochastic process*. The stochastic process takes the form $SP : \Omega \times \mathbb{N} \rightarrow E$, but the common definition of a stochastic process is sufficient for the discussion in this section. A *reward variable* is a mapping $Y : SP \rightarrow \mathbb{R}$.

While this definition is nice because of its generality, it lacks any structure that would allow us to evaluate Y (efficiently) in practice. Researchers commonly refine the the definition of a reward variable in a number of ways to make evaluation more efficient. First, the reward variable is defined on sample paths of SP. That is, given a particular outcome $\omega \in \Omega$, the stochastic process simply becomes $SP|\,\omega : \mathbb{R} \rightarrow \mathbb{R}$. This simplifies reward variables because it can be defined in terms of a one-dimensional function. Defining the function $SP|\,\omega$ is still non-trivial. For example, Markov-chain-based solutions do not operate on sample paths;

rather, they solve in terms of distributions over outcomes. Further refinements are necessary for efficient Markov-chain-based solutions.

We note that the stochastic process is generally constant for intervals of times, and not continuous at the interval boundaries. This is the nature of discrete event systems: a system is in a state for some time, and then, at a discrete point in time, the system changes state. We can generalize then that most useful reward variables are functions of the states the system is in, how long the system stays in each state, and/or the jumps between states. These "measures" (we use the term "measure" informally) are sufficiently captured using "rate" and "impulse" rewards, which we discuss below.

Next, we can observe that reward variables are typically of interest only at some instant or interval of time. For example, we may want to know the steady-state instant-of-time availability of a system. The time at which we are interested in the reward variable (the "utility time") is an instant of time t taken as $\lim_{t\to\infty}$. Thus, we need only "accumulate" or "observe" reward over certain intervals or at certain instants of time.

Finally, it is not always desirable or practical to solve for the reward variable in distribution. It may be the case, for example, that the modeler wishes to know only the mean (e.g., computing reliability), or $\Pr[Y > k]$ (for risk assessment). Ultimately, what is desired is some measure of Y called a *Y-measure* or *reward variable measure*.

We review the work of others in the area of reward variable specification in Section 4.2. We describe the approach taken in Möbius in Section 4.3; specifically, we discuss the formal definitions, and how we address the utilization time and reward variable measure. In Section 4.4, we demonstrate how to calculate Möbius reward variables, and illustrate with some examples in Section 4.5. We conclude in Section 4.6.

4.2 Related Work

The majority of modern reward variable specification techniques are based on work of [67], which is based on reward models of Markov or semi-Markov processes. *Reward*, which is some unitless quantity used in calculating a reward variable, is accumulated in one of two ways: it can be accumulated at a certain rate as long as the model is in a particular state or set of states, or it can be accumulated in impulses when the model transitions between states. Reward is summed over some interval, or observed at some instant, and the result is the reward variable.

Based on this, formalisms usually include what we call a reward structure, which is made up of two functions: one for rate reward and one for impulse reward. Most reward formalisms

have something similar to that, although some earlier reward formalisms did not have impulse rewards. A *rate reward* function specifies the rate at which reward is accumulated for any state, and an *impulse reward* function specifies the amount of reward accumulated when the model changes state.

Just as there are many atomic modeling formalisms for specifying model behavior, there are many reward formalisms for measuring model behavior. We choose to focus on three modern formalisms: those used for SRNs [66], SANs [45], and path-based reward variables [42]. We believe these to be representative of the state-of-the-art in reward specification. Again, the Möbius framework needs to have a reward variable specification that is general enough to accommodate the various reward variable formalisms and extensions likely to be developed in the near future.

We show in this chapter that many of the pre-existing concepts that we review have been extended and generalized in Möbius. For example, we extend the notion of an event to include the sojourn time; this allows, for example, impulse rewards to be a function of the state, the action that completes, and the sojourn time. While this technically eliminates the necessity for rate rewards, we retain them for the sake of both efficient solution and straightforward formalism-to-framework translation, and for historical reasons.

The Möbius reward variable also includes relevant information about itself that is not always (directly) associated with reward variables as implemented in various formalisms. For example, Möbius reward variables include utility time and the reward variable measure. Utility time is the time the reward variable is measuring the stochastic process. Thus, the reward variable is independent of the behavior of the stochastic process for time outside of the utility time. For example, if a reward variable measures the interval availability of a model from time 0 to time t, then the utility time would be $[0, t]$. The behavior of the stochastic process at time (t, ∞) would not affect the reward variable. Typically, formalisms leave the utility time unspecified and require the user to specify it when performing a solution. This may at first seem unusual, but there is a reason for this. The utility time is often set when a solver begins the solution process, and the same utility time is used for solving all the reward variables. (Solving all reward variables for the same utility time is usually not significantly more computationally expensive than solving for one reward variable.) While convenient for tool builders, this is conceptually inefficient in that the information about the reward variable is specified in two separate places. We believe that the utility time should be specified as part of the reward variable. It is possible for solvers to override the utility times and compute different utility times if the solver permits it and the user desires it, so no generality is lost.

Möbius also includes a reward variable measure as part of the reward variable definition.

A reward variable measure indicates what information we want to compute with regards to a reward variable. Again, the reward variable measure is typically determined by the solver, and all reward variables are computed with the same reward variable measure. Möbius takes a different approach. For example, the reliability of a system can be computed as the expected value of a reward variable. Computing the variance of this reward variable is meaningless, but many solvers will do it anyway.

The utility time and reward variable measures are absent from the SAN reward structures and the path-based reward variables. In SRNs, the utility time is assumed to be $[0, \infty)$, and the reward variable is the mapping $Y : SP \rightarrow \mathbb{R}$ discussed in the introduction. As we discuss below, there is no support for expressing the nature of the function; we find this unsatisfactory for a framework.

We have studied other model measurement approaches, including that found in the model checking community. In particular, we examined continuous stochastic logic (CSL) [68], [69], [70], an extension to computational tree logic (CTL) [71]. We concluded that the approach taken by reward variables and CSL are fundamentally different. Recall that reward variables take the form $Y : SP \rightarrow \mathbb{R}$. CSL is constructed by operators of the form $f : 2^{\Sigma} \rightarrow 2^{\Sigma}$. Operators of f are repeatedly applied, and after each operator is applied, some Boolean property of the model may be checked. The last operator applied may be evaluated as a probability as well as a Boolean property. For example, an operator may be able to answer the question "Does the system have reliability 0.99?" while the last operator could answer the question "What is the system reliability?"

Through our studies and discussions, we have determined that the two classes of measure specification have considerable overlap, but that neither is a superset of the other. We are aware of research efforts under way trying to address this issue. However, at the time of this writing, unfortunately, we know of no measure specification that can capture both model-checking techniques and reward variables. We believe reward variables are better suited for performance and dependability evaluation, the primary purpose of Möbius, and that model checking is better suited for model validation, which is of secondary importance to Möbius. Thus, we use reward variables in Möbius.

4.3 Möbius Approach

The approach we took for Möbius most closely resembles that of path-based reward variables, but with many significant extensions. The primary differences provided by Möbius are that the utilization time can be determined by the model (i.e., is a function of the model),

and that impulse rewards may be a function of the state sojourn time. Also, as previously mentioned, Möbius reward variables include the utility time and reward variable measure as part of the reward variable specification.

We begin by describing the Möbius reward variable. The reward variable is made up of five basic structures: the reward automata, the reward structure, the reward control, the reward variable measures, and reward properties. Briefly, the automata is similar to the automata of path-based reward variables in that it allows for the measurement of sequences of events. The reward structure is also similar to SAN reward variables, and has been extended much like path-based reward variables to accommodate the automata. The reward control is unique to Möbius reward variables and specifies various parameters to the reward variable, such as whether the reward variable is an instant-of-time variable, interval-of-time variable, or time-averaged interval-of-time variable, and specifies the utilization time in terms of an absolute time or in terms of events in the model. The reward variable measures indicate which measure of the reward variable (a random variable) the solvers should solve. Reward properties are any extra information that may aid in the efficient solution of reward variables, or may be necessary to enable the use of specialized solvers.

Before we describe reward variables formally, we introduce some notation. We use the notation $[(a, b])$ to mean any of the open, closed, or semi-closed intervals defined by the endpoints a and b, i.e., $[a, b]$, $[a, b)$, $(a, b]$, or (a, b). We use this notation when the details of one or both endpoints are not relevant to the immediate discussion. Let I be an interval of the form $[(a, b])$ for $b > a$, or $[a, a]$. We define $\mathcal{B}_{\mathrm{f}}(\mathbb{R}) = \{I_1, \ldots, I_n\}$ to be a finite, nonoverlapping set of intervals over $\mathbb{R}$ such that $I_j \cap I_k = \emptyset, \forall j, k$. Note that $\mathcal{B}_{\mathrm{f}}(\mathbb{R})$ is the subset of the Borel measurable set that can be expressed as a finite set of intervals. Since the intervals must be enumerated explicitly by a computer program or file, we restrict ourselves to a finite set of intervals.

4.3.1 Automata

Automata are used for measuring sequences of events, sometimes called *paths*. (For several examples, see [41], [42].)

The *automata of a Möbius reward variable*, also called the *reward automata*, includes the following elements.

$rt \in T$, the type of the reward variable automata state (similar to the type of a state variable). See Section 3.2.1 for a description of T.

$\delta : rt \times E \rightarrow rt$, the automata transition function, where for a automata state $\rho \in rt$ and

an event $e \in E$, $\delta(\rho, e)$ determines the next state of the reward variable automata.

The *reward automata functions* are given as

$$RA : R \rightarrow rt \times \delta \,.$$

The state of the automata is given by the mapping

$$\rho : R \rightarrow rt \,,$$

where for $\rho(r) \in rt$, rt is the type of the reward variable automata state for reward variable r. We often simply write ρ when there is no ambiguity about which reward variable automata state we are referring to.

4.3.2 Reward structure

The *reward structure of a Möbius reward variable* is made up of the following elements.

$\mathcal{C} : rt \times E \rightarrow \mathbb{R}$, the *impulse reward function*, where for all reward variable automata states $\rho \in rt$ and events $e \in E$, $\mathcal{C}(\rho, e)$ is the impulse reward earned when the reward variable automata is in state ρ and event e occurs.

$\mathcal{R} : rt \times \Sigma \rightarrow \mathbb{R}$, the *rate reward function*, where for all reward variable automata states $\rho \in rt$ and model states $\sigma \in \Sigma$, $\mathcal{R}(\rho, \sigma)$ is the rate at which reward is earned when the reward variable automata is in state ρ and the state of the state variables of the model is σ.

The *reward structure functions* are the mapping

$$RS : R \rightarrow \mathcal{C} \times \mathcal{R} \,.$$

This definition is similar to that of SANs and SRNs, but extended to take advantage of the automata, as path-based reward variables do, and our notion of an event, which includes the sojourn time.

4.3.3 Reward control

Reward control determines how and when to collect reward. Most of the reward control functions deal with the utility time.

The *reward control functions* of a Möbius reward variable include the following elements.

RVType : $\{Instant, Interval, TAInterval\}$, the reward variable type that indicates whether the reward variable is an instant-of-time variable, interval-of-time variable, or a time-averaged interval-of-time variable.

UtilityType : $rt \times \Sigma \rightarrow \{Fixed, Variable\}$, indicates whether the start of the interval is given by a set of intervals (given by *UtilityTime*) or by events (indicated by other reward control functions).

UtilityTime : $\mathcal{B}_{\mathrm{f}}(\mathbb{R}^{\geq})$, indicates length of the utilization time interval for interval-of-time and time-averaged interval-of-time variables. An interval time of ∞ is taken to be a limiting value. For instant-of-time variables, this set is required to take the form $\{[a, a], \ldots\}$, where $a = \infty$ is taken to mean the steady-state instant-of-time. For interval variables, this set takes the form $\{([a, b)], \ldots\}$, where $a \leq b$.

StartType : $rt \times \Sigma \rightarrow \{Timed, Event\}$, indicates that the beginning of the next utility interval is given by a fixed amount of time (*Timed*) or by a subsequent event (*Event*).

StartTime : $rt \times \Sigma \rightarrow \mathbb{R}^{\geq} \cup \{\infty\}$, for reward variables with a *StartType* of *Timed*, this indicates how long to wait after the end time of the previous interval, or time zero, before starting a new utility time interval. A *StartTime* of ∞ is taken to mean $\lim_{t \rightarrow \infty}$.

StartEvent : $rt \times E \rightarrow bool$, indicates the start of the utilization time interval when the function first becomes *true* if the reward variable is an interval-of-time or time-averaged interval-of-time variable, or the time instant if it is an instant-of-time variable. The start of the interval is closed if $Closed = true$; otherwise, it is open.

Closed : $rt \times \Sigma \cup E \rightarrow bool$, indicates whether the start or end of the interval is open or closed for intervals of the *Event* type.

EndType : $rt \times \Sigma \rightarrow \{Timed, Event\}$, indicates whether the end of the interval is determined by *EndTime* or by an event indicated by *EndEvent*.

EndTime : $rt \times \Sigma \rightarrow \mathbb{R}^{\geq} \cup \{\infty\}$, indicates the duration of the interval if the interval type is *Timed*.

EndEvent : $rt \times E \rightarrow bool$, is used to indicate the end of the interval when the end type is determined to be *Event*.

EndUtility : $rt \times \Sigma \rightarrow bool$, indicates when the utility time intervals are completed for *Variable* reward variables.

The *reward control functions* is the mapping

$$RC : R \rightarrow RVType \times UtilityType \times UtilityTime \times StartType \times StartTime \times StartEvent \times Closed \times EndType \times EndTime \times EndEvent \times EndUtility.$$

4.3.4 Computing utility time

The computation of the utility time is somewhat complicated. Figure 4.1 illustrates the algorithm graphically. The utility interval can be determined in two different ways. If $UtilityType = Fixed$, then the utility time is given completely by $UtilityTime$ as a finite set of disjoint intervals. If $UtilityType = Variable$, then the utility time may be determined as a function of the evolution of the model. This involves evaluating a number of functions outlined below.

First, $StartType$ is used to determine whether the instant, or the start of the time interval, is at a fixed time point, or given by some event. At time zero, this is used to determine the start of the first interval. Otherwise, it is used to determine the start of the next instant or interval relative to the end of the most recent instant or interval. Let t_0 be the end of the last interval, or 0 if no previous interval exists. Let the state at time t_0 be σ_{t_0}, and the reward variable's automata state be ρ_{t_0}. If $StartType = Timed$, then the start of the next interval, t_1, is given by $t_1 = t_0 + StartTime(\rho_{t_0}, \sigma_{t_0})$. Let σ_{t_1} be the state of the state variables at time t_1. If σ_{t_1} is not unique, then let Σ_{t_1} be the set of states that the model takes at time t_1. Then we define $\sigma^*_{t_1} = \max_i\{\sigma_i : \sigma_i \in \Sigma_{t_1}\}$, i.e., $\sigma^*_{t_1}$ is the most recent state at time t_1. Let $\sigma_{t_1} = \sigma_{t_1}*$, and ρ_{t_1} be the automata state corresponding to σ_{t_1}.

If $StartType = Event$, then the next interval is determined by the first event that occurs at or after t_0 in which $StartEvent(\circ) = true$. Call this time t_1, and the event $e_{t_1} = (\sigma_{t_1}, \tau, a, \sigma')$. Thus, we use t_1 as the start of the next interval, which is independent of the value of $StartType$.

If the reward variable type is an instant-of-time variable, the instant of time is t_1. Next, $EndUtility$ is evaluated to determine if the utility time is complete. If $EndUtility(\rho_{t_1}, \sigma_{t_1}) = true$, then the utility time is complete; otherwise, the algorithm starts over.

If the reward variable type is an interval-of-time or time-averaged interval-of-time variable, then at the beginning of the utility interval, $Closed(\rho_{t_1}, \sigma_{t_1})$ is evaluated. Otherwise, if $StartType = Event$, then $Closed(\rho_{t_1}, e_{t_1})$ is evaluated. If $Closed(\circ) = true$, then the interval contains t_1, i.e., $I = [t_1, t_2]$). Otherwise, the interval excludes t_1, i.e., $I = (t_1, t_2]$).

The end of the interval is determined in one of two different ways. First, if $EndType(\rho_{t_1},$

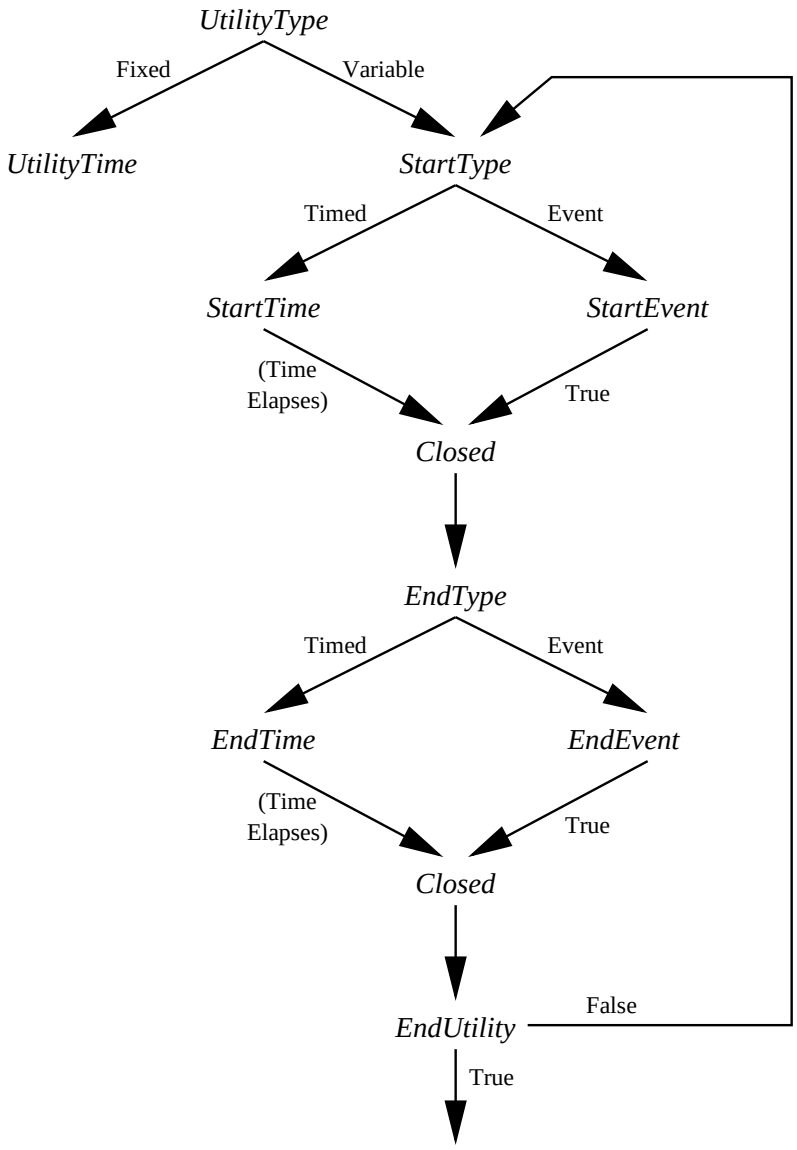

Figure 4.1: Flowchart of function evaluation for interval reward variables.

$\sigma_{t_1}) = \mathit{Timed}$, then the end of the interval is a fixed time larger than t_1 and given by $\mathit{EndTime}$. The end of the interval is calculated by $t_2 = t_1 + \mathit{EndTime}(\rho_1, \sigma_1)$. Similarly, we let σ_{t_2} be the state at time t_2; if σ_{t_2} is not unique, then let Σ_{t_2} be the set of states at time t_2, let $\sigma_{t_2} = \sigma^*_{t_2} = \max_i\{\sigma_i : \sigma_i \in \Sigma_{t_2}\}$, and let ρ_{t_2} be the automata state corresponding to σ_{t_2}. Otherwise, if $\mathit{EndType}(\rho_{t_1}, \sigma_{t_1}) = \mathit{Event}$, then the next interval is determined by the first event that occurs at or after t_1 in which $\mathit{EndEvent}(\circ) = \mathit{true}$. Let the time at which this event occurs be called t_2, let the event $e_{t_2} = (\sigma_{t_2}, \tau, a, \sigma')$, and let ρ_{t_2} be the automata state corresponding to σ_{t_2}.

The end of the interval is open or closed, depending on the value of *Closed*, at time t_2.

If $EndType(\rho_{t_1}, \sigma_{t_1}) = Timed$, and if $Closed(\rho_{t_2}, \sigma_{t_2}) = true$, then the interval contains t_2; otherwise, it excludes t_2. If $EndType(\rho_{t_1}, \sigma_{t_1}) = Event$, then $Closed(\rho_{t_2}, e_{t_2})$ is evaluated to determine whether t_2 is included in the interval.

Finally, $EndUtility(\rho_{t_2}, \sigma_{t_2})$ is evaluated. If it evaluates to *true*, then no more intervals or instants exist in the utility time. Otherwise, *StartType* is evaluated to determine how to compute the time until the next start of an interval, and the process repeats.

This algorithm describes how the utility time can be a function of the evolution of the model. Each "loop" through the algorithm defines a separate time interval or instant, and the utility time is the union of the intervals. Note that this algorithm allows a set $U = \{(a, b][b, c]\}$ to be defined (b is included twice). As stated earlier, the utility time intervals must be disjoint, so U is invalid.

4.3.5 Reward variable measures

While a reward variable defines a measure on the underlying stochastic process of the Möbius model, sometimes the desired solution to the reward variable is the complete distribution of the reward variable. However, sometimes computing the complete distribution is difficult or impractical, or simply not desired. For example, for a reliability measure it is sufficient to compute only the expected value of a reward variable. That is why the Möbius reward variable includes the reward variable measure as part of the reward variable specification.

The reward variable measure indicates the desired measure of the reward variable, which is a random variable. This is useful for several reasons. First, some reward variable formalisms include this information as part of the reward variable specification, and this information has been found useful in CSL. Also, since some reward measures are costly to compute for each reward variable, users may wish to assign reward variable measures differently for different reward variables.

The *reward variable measures* of a Möbius reward variable include the following elements.

Dist : *bool*, the indicator variable that indicates whether the solver should solve for this reward variable in distribution.

Moments : $\mathbb{Z}^+$, a set of positive integers indicating the set of moments for which the solvers should solve.

Intervals : $2^{\mathcal{B}_{\mathfrak{f}}(\mathbb{R})}$, a finite set of interval sets; the solver should calculate the probability that the reward variable lies in U for each $U \in Intervals$.

Note that in many ways, reward variable measures are like properties. For example, new types of reward variable measures may be added without invalidating existing solvers. If a solver encounters a reward variable measure that it does not understand, it may safely ignore it. We anticipate, however, that the reward variable measures we give are adequate for most needs and satisfy the needs of all the solvers we are currently aware of.

The *reward variable measures functions* of a Möbius reward variable is given by

$$RMeas : R \rightarrow Dist \times Moments \times Intervals\,.$$

4.3.6 Reward variable properties

Each reward variable also has an associated set of properties. Properties may give useful hints to solvers to aid in efficiency of solution. For example, several reward variables may share the same reward automata functions and have identical state. A property could state this, eliminating the need for redundant reward automata.

Let $RP : R \rightarrow 2^{\Pi}$ be the reward variable properties.

One example of an important reward property is the <Markov> property. The <Markov> property holds if all the reward variable functions that are functions of events are independent of the sojourn time. Solution methods based on numerical solutions of Markov chains will require as a condition of their use that reward variables have the <Markov> property.

4.3.7 Reward variable definitions

The *reward variable functions* of a Möbius model is the tuple

$$RF = (RA, RS, RC, RMeas, RP)\,,$$

where RA is the reward automata, RS is the reward structure, RC is the reward control, $RMeas$ is the reward variable measure, and RP is the reward variable properties.

The *reward variable state* is the state of the reward variable automata ρ. Formally, the state of the reward variable automata is given by

$$\rho : R \rightarrow rt$$

such that $\rho(r) \in r.rt$ (where $r.rt$ is the rt element of $RA(r)$). We often write simply ρ if the reward variable r is irrelevant or clear from context. The initial automata state of a reward variable, ρ_0, is given explicitly.

4.3.8 Reward model

A reward model is an atomic model with associated reward variable information. One element not yet covered is reward model properties. These are any properties that apply to the reward model as a whole. Let $RMP : R \to 2^{\Pi}$ be the reward model properties.

Formally, a *reward model* is the tuple

$$RM = (AM, R, RF, \rho_0, RMP)\,.$$

Note that a reward model is an augmented atomic model. A reward variable formalism takes an atomic model, adds reward variable information, and results in a reward model. The reward variable information is simply a set of reward variables (names), all the associated functions of the reward variables, the initial automata state, and any reward model properties.

Note that we can trivially translate an atomic model into a reward model if $R = \emptyset$. This may be useful if, for example, we wish to compose atomic models. This is discussed more in Chapter 7.

4.4 Reward Variable Computation

Following the approach of [45], we can give expressions for the value of a reward variable. First, we introduce some notation. Let E' be a homomorphism of E where the $e \in E$ is mapped to $e' \in E'$ by the function such that $\forall \tau \in \mathbb{R}^{\geq}$, $(\sigma, \tau, a, \sigma') \mapsto (\sigma, \circ, a, \sigma')$. One can think of E' as the set of "untimed" events. The set E' is a useful abstraction of E if the reward variable is Markov, which is the scenario we consider first. We call e' an "untimed event." Note the distinction: I is an interval, while $\bar{I}$ is an indicator random variable.

We begin by defining some random variables that can be evaluated from the model that has the Markov property and which we use to define our reward variable expressions.

- $\bar{I}^t_{(\rho,\sigma)}$ is an indicator random variable representing the event in which the state of the reward variable automata is ρ and the state variables of the model are in state σ at time t.

- $\bar{I}^t_{(\rho,e')}$ is an indicator random variable representing the event in which for $e' = (\sigma, \circ, a, \sigma')$, action a completes at time t in state σ while the reward variable automaton is in state ρ, resulting in state σ'.

- $\bar{J}^I_{(\rho,\sigma)}$ is a random variable representing the total time during the interval I that the

reward variable automata are in state ρ and the model state variables are in state σ.

- $\bar{N}^I_{(\rho,e')}$ is an indicator random variable representing the total number of times within the interval I that the reward variable automata are in state ρ and untimed event e' occurs.

Continuing with this approach, we can define the various reward variables according to the reward variable type. The following equation defines the instantaneous reward variable at time t. Let $U = \{I_i\}$ be the utility time, where I_i takes the form of $[a, a]$.

$$\bar{V}_U = \sum_{[t,t]\in U} \left(\sum_{(\rho,\sigma)\in rt\times\Sigma} \mathcal{R}(\rho,\sigma)\bar{I}^t_{(\rho,\sigma)} + \sum_{(\rho,e')\in rt\times E'} \mathcal{C}(\rho,e)\bar{I}^t_{(\rho,e')} \right)$$

Note the following things about this equation. The stochastic process is right-continuous, and events may occur in zero time. Thus, it is possible to accumulate multiple rate and impulse rewards at an instant of time. Also, E' is discrete space, and therefore countable; thus, we are able to sum over the space. Furthermore, even though rt may not be countable, ρ may take on only a countable number of values, so we can sum over the countable subset of rt that (ρ, e') may take on.

The interval-of-time reward variable can be defined using a similar approach. Let $U = \{I_i\}$ be the utility time of the reward variable.

$$\bar{Y}_U = \sum_{I\in U} \left(\sum_{(\rho,\sigma)\in rt\times\Sigma} \mathcal{R}(\rho,\sigma)\bar{J}^I_{(\rho,\sigma)} + \sum_{(\rho,e')\in rt\times E'} \mathcal{C}(\rho,e')\bar{N}^I_{(\rho,e')} \right)$$

The intuition for an instant-of-time reward variable is that it is similar to

$$\lim_{\Delta t\to 0} \frac{\bar{V}_{\{[0,t+\Delta t]\}} - V_{\{[0,t]\}}}{\Delta t}\,.$$

Since the stochastic process is right-continuous, this limit exists. However, this equation ignores all impulse rewards. Rather than eliminate the impulse reward component of instant-of-time reward variables (which can be set to zero), we chose to interpret instant-of-time impulse rewards as the probability that some event e occurs at exactly the measured time instant.

Finally, the time-averaged interval-of-time variable can be computed simply as

$$\bar{W}_U = \frac{\bar{Y}_U}{|U|}\,,$$

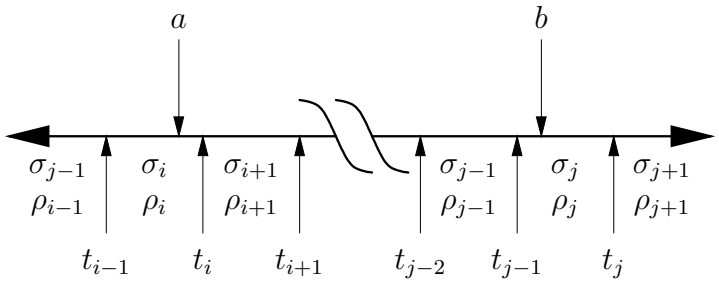

Figure 4.2: Reward variable computation for a sample path.

where $|U| = \sum_{I \in U} |I|$, and for $I = [(a, b])$, $|I|$ is taken to mean $b - a$.

In general, it is more difficult to express non-Markov reward variables compactly because it involves performing a countably infinite number of integrations over time intervals. However, we are able to define them in terms of a sample path. Note that since the state sojourn time is a part of the Möbius stochastic process, there is an uncountable number of sample paths.

Let $e_0, e_1, \ldots$ be a sample path, where $e_i = (\sigma_i, \tau_i, a_i, \sigma_{i+1})$, and ρ_i is the automata state of the reward variable when the model is in state σ_i. The instant-of-time reward variable is computed as

$$V_U = \sum_{i=0}^{\infty} \sum_{I \in U} \mathcal{R}(\rho_i, \sigma_i) I_{[t_{i-1}, t_i) \cap I \neq \emptyset} + \mathcal{C}(\rho_i, e_i) I_{[t_i, t_i] = I}$$

where $t_{-1} = 0$ and I_e is 1 if e is true, 0 otherwise.

Next, we describe the computation of interval-of-time reward variables. Figure 4.2 illustrates this for one interval.

$$Y_I = \sum_{i=0}^{\infty} \mathcal{R}(\rho_i, \sigma_i) |[t_{i-1}, t_i) \cap U| + \mathcal{C}(\rho_i, e_i) I_{[t_i, t_i] \in U}$$

Finally, the time-averaged interval-of-time reward can be computed.

$$W_U = \frac{Y_U}{|U|}$$

4.5 Examples

4.5.1 SRNs

We begin with comparing the Möbius reward variable to the measure specification for SRNs. One element of an SRN is the $M = \{(\rho, r_i, \psi_i)\}$, a finite set of measures. (For

consistency, we use the SRN notation when referring to SRNs.) The term ρ is similar to the rate reward $\mathcal{R}$, and r is similar to the impulse reward $\mathcal{C}$. In SRNs, the rate and impulse rewards are *always* accumulating reward according to a function Y, which is given as a function of time.

$$Y(t) = \int_0^t \rho(\mu(u))du + \sum_{j=1}^{max\{n:t_n \leq t\}} r_{t_j}(\mu^{[j-1]}),$$

where $\mu(t)$ corresponds to the state at time t and is similar to our σ_t notation, and $\mu^{[i]}$ corresponds to σ_i, which is the state corresponding to the ith event. (Note the use of the subscript i on σ as a sequence number and on t as a time.)

The element ψ is a function that maps the stochastic process Y to a real number. The codomain is the "answer" to the reward variable. This approach is significantly different from the Möbius approach, which seeks to expose the function using automata and utility times. We chose our approach because the Möbius tool needs to capture and express what the function is, and do it in such a way that solvers can understand, manipulate, and calculate it. This results in significantly more structure and a decrease in generality. We believe this structure is necessary for efficient solution by tools. Using a general function such as ψ would require solvers to compute a complete characterization of the stochastic process, which is often unnecessary.

We continue by illustrating some examples of SRN measures found in [66].

- Expected number of transition firings up to time t. Let $\mathcal{R}(\circ) = 0, \mathcal{C}(\circ) = 1$, the reward variable type be interval-of-time, the utility type be fixed, the utility time be $[0, t]$, and the reward variable measure compute the first moment.

- Expected time-averaged reward up to time t: $E\left[\frac{Y(t)}{t}\right]$. This is done in Möbius using a time-averaged interval-of-time variable with the appropriate rate and impulse rewards, the utility type fixed, utility time $[0, t]$, and the reward variable measure computes the first moment.

- Expected instantaneous reward rate at time t: $E\left[\lim_{\delta \to 0} \frac{Y(t+\delta)-Y(t)}{t}\right]$. This is done with an instant-of-time variable with a fixed utility type and the utility time of $[t, t]$.

- Expected accumulated reward in the steady state: $E\left[\lim_{t \to \infty} Y(t)\right]$. This is done as above, except that the utility time is $[0, \infty)$.

- Mean time to absorption. This can be expressed by a reward variable with a rate reward of 1 if the state is transient or 0 if it is absorbing, and an impulse reward of 0. All else is as above.

- Expected instantaneous reward in steady state:

$$E\left[\lim_{t\to\infty}\lim_{\delta\to 0}\frac{Y(t+\delta)-Y(t)}{\delta}\right].$$

This is done with an instant-of-time variable with a fixed utility type, and the utility time uses the special interval $[\infty,\infty]$, which is taken to mean $\lim_{t\to\infty}[t,t]$. The reward variable measure computes the first moment. Under certain conditions, this measure can also be computed as the expected time-averaged reward in the steady-state: $E\left[\lim_{t\to\infty}\frac{Y(t)}{t}\right]$. This can also be expressed using a time-averaged interval-of-time variable with a fixed utility time of $[0,\infty)$; the reward variable measure computes the first moment.

4.5.2 SANs

We borrow significantly from the work of [45], especially in notation. SAN reward variables are made up of a reward structure, which is similar to the Möbius reward structure, a type: instant-, interval-, or time-averaged interval-of-time, again similar to Möbius, and a utility time of $[t,t]$ for instant-of-time variables or $[t,t+l]$ for interval-of-time or time-averaged interval-of-time variables.

SAN reward variables are a subset of Möbius reward variables. This is obvious from the similarities in notation. It is easy to construct a proof by observing the reward variable calculations. Möbius reward variable calculations $\bar{V}_U$, $\bar{Y}_U$, and $\bar{W}_U$ are supersets of the SAN reward variable calculations of $\bar{V}_I$, $\bar{Y}_I$, and $\bar{W}_I$.

Möbius reward variables have three significant extensions. First, reward variables have automata, which allow Möbius reward variables to measure sequences of events. This is similar to the work of Obal [42]. Another extension is that Möbius reward variables have more flexible utility times, including open and closed intervals, multiple intervals or instants, and intervals and instants that are functions of the execution of a model. Finally, the Möbius event includes the sojourn time, and for Möbius reward variables without the <Markov> property, reward variables may also be functions of the sojourn time.

4.5.3 Path-based reward variables

Path-based reward variables [41], [42] are similar to SAN reward variables with the extension of an automaton for each reward variable. Therefore, in general terms, our remarks on SANs also holds for path-based reward variables, except regarding the automata.

One difference between Möbius reward variables and path-based reward variables is that for path-based reward variables, the path automaton has a set of (possibly empty) final states. The Möbius reward variable automata do not have a set of final states. This is important because the stopping times of path automata are determined using those final states. Path automata do not have the reward control information explicitly associated with the reward variable, so direct comparison is difficult. However, we use the various reward variable expressions of [42] for comparisons.

There are two types of interval utilization times for path-based reward variables: those of the form $[t, t+l]$ where t and l are constants, and those of the form $[t, T_F]$, where T_F is the time for the automata to reach a final state. It is easy to implement intervals of the type $[t, t+l]$ in Möbius by using a fixed interval type. Intervals of the type $[t, T_F]$ can be implemented using the same automata, a fixed start time (t) and a variable interval time with $\mathit{EndEvent}(\rho, e) = \mathit{true}$ when ρ enters a final state. Thus, while the automata may behave slightly differently, the differences occur after the utility time, and Möbius reward variables are able to measure the same behaviors that path-based reward variables measure.

Note the inherent difference between interval utility times of Möbius and path-based reward variables. Path-based reward variables have interval utility times that start at a fixed time instant, while Möbius allows events to trigger the start of the interval. Path-based reward variables can mimic this behavior for interval-of-time variables by adding extra structure to the model and masking the rate and impulse rewards, but it is not possible to achieve an equivalence with time-averaged interval-of-time variables. Möbius also allows the utility time to be broken up into multiple intervals, while path-based reward variables only allow one interval. Again, path-based reward variables can mimic multiple time intervals by modifying the model and masking the rate and impulse rewards, but no equivalence exists for time-averaged interval-of-time variables.

4.5.4 Equivalence classes

It is frequently possible to modify or augment a model and use the SAN rate- and impulse-based reward variables to obtain the same measures that are possible with Möbius reward variables. In this section, we examine several classes of Möbius reward variables, including some that can be expressed with SAN reward variables with an augmented model, some that can be expressed with path-based reward variables with an augmented model, and some that can not be expressed except in Möbius. Finally, we argue that the Möbius approach is "better" than other approaches. We offer only informal arguments.

First, we note that SAN reward variables implement instant-of-time impulse rewards dif-

ferently from most formalisms and Möbius. An instant-of-time SAN impulse reward defined at time t does not measure the activity completion at time t, but rather the last activity to complete before or at t. The Möbius reward variable automata can be modified to reflect the last measurable action completion, so SAN instant-of-time impulse rewards can be captured in Möbius. The reverse is not true.

We begin by comparing instant-of-time reward variables. Let $U = [t, t]$ be the utility time. If t is a fixed time unit, then SAN reward variables behave much like Möbius reward variables, with the exceptions noted above. If the utility time is based on an event, then SAN reward variables and path-based reward variables are not able to capture this behavior. Finally, if $U = [t_1, t_1], [t_2, t_2]$, i.e., the utility time contains more than one time instant, then neither SAN nor path-based reward variables are able to capture this behavior.

Interval-of-time reward variables have a larger equivalence class. First, we note that it is possible to mimic the Möbius reward variable automata in a model by augmenting the model with a zero-timed action and a state variable. The zero-timed action implements the δ function, while the state variable implements the automata state. The mimic is not perfect, as zero-timed actions can change state only as a function of the state and not as a function of events. Since the utility time of a reward variable acts as a "mask" for the rate and impulse rewards, we can argue that there always exists a rate and impulse reward with a utility time of $U = [0, \infty)$ that defines the same reward variable. Note that this may destroy the Markov property of the model. If any of the start or end points of the time intervals are fixed, the model must be augmented with fixed-time actions to mimic this behavior, and fixed-time actions are inherently not Markovian. However, we believe it is frequently, although not always, possible to augment a model and use SAN reward variables to implement Möbius instant-of-time reward variables.

Time-averaged interval-of-time reward variables have a much smaller equivalence class. As we illustrated earlier, it is frequently possible to capture interval-of-time reward variables by augmenting the model. It is also possible to create a reward variable that measures the utility time of a reward variable. However, it is not possible in general to know the correlation between the two. For example, to calculate the expected time-averaged interval-of-time variable, one can calculate the expected interval-of-time variable (provided it is finite) and the expected utility time and divide the expected interval-of-time measure by the expected utility time measure. However, it is not possible to calculate higher moments, the probability that the reward variable lies in an interval, or the distribution. Only time-averaged interval-of-time reward variables with utility times of the form $[a, b]$, where both a and b are constants, can be computed with SAN reward variables; only time-averaged interval-of-time reward variables with utility times of the form $[a, b]$, where a is constant and

b is either constant or based on events, can be computed with path-based reward variables.

We argue (informally) that Möbius reward variables are "better" than the reward variable formalisms we have reviewed for a number of reasons. First, all the information necessary to express and calculate a reward variable is encapsulated within the reward variable. It is not necessary to augment the model in order to measure something about the model. This is conceptually pleasing, and, as argued in [42], can aid in efficient numerical solution of Markov models. The efficiency arises from expanding the state space of Markov models selectively and intelligently based on the path automata.

In general, it is also easier to translate formalism reward variables into Möbius reward variables than the reverse. While the reverse is sometimes true, it is tedious and could require extensive augmentation of the model. Translation of formalism reward variables into Möbius reward variables, although not trivial, requires only minor changes, and the changes are only to the particular reward variable being translated.

Finally, the Möbius reward variable (unlike the SRN approach) has sufficient structure to allow solvers to operate efficiently, whether they are simulators or analytic solvers. At the same time, the Möbius reward variable is general enough to capture most interesting behaviors, and certainly more behaviors than is possible to capture with SANs and path-based reward variables.

4.6 Conclusion

Reward variables are an important part of the modeling process because they allow users to learn information about the models they have built. Reward variables are perhaps the most complicated major component of the framework. This complexity arises from our desire to have a great deal of flexibility and expressibility, but still retain sufficient structure so that solvers can compute solutions efficiently.

Möbius reward variables are a significant generalization of formalism reward variables. In particular, Möbius reward variables allow us to capture the rate and impulse rewards of many formalisms. The automata allow us to capture path-based reward variables as well. Furthermore, we argued informally that Möbius reward variables are a superset of various formalism reward variables.

We have further extended Möbius reward variables by including an explicit utilization time and reward variable measure. This allows users of formalisms to express exactly what they want to know about reward variables, and localizes the information about reward variables to the reward variable structure.

CHAPTER 5

FORMAL MÖBIUS EXECUTION POLICY

5.1 Introduction

All the formal definitions of the various Möbius reward model components are present in order to define a stochastic process that we can then measure. In this chapter, we formally define the stochastic process. Previously, we said that the stochastic process of the model is given by a sequence of events, where an event is the 4-tuple including a state, the sojourn time, the action that completes, and the resulting next state. The "state" is taken to mean the state of the state variables.

This definition is useful in describing the evolution of the model for reward variables, and hence we call it the *measurable stochastic process*, or the *stochastic process*. However, it is not sufficient for describing what the next event will be given the current event. The next event can be determined by the entire sequence of previous events, but this is inefficient for our needs.

What we can do is augment the notion of "state" to include the action state and reward variable state. Using this notion of an augmented state, we are then able to describe the next event given the current event. Using the augmented state, we can define an *augmented event*, which includes the augmented state, sojourn time, and completing action. A sequence of augmented events makes up the *augmented stochastic process*. The augmented stochastic process can be classified as a time-homogeneous generalized semi-Markov process [52], [53]. To be clear, we use the symbol ς to be this "augmented state," and ε to be the event using the augmented state. We define each of these formally, and then formally define the Möbius stochastic process in the remainder of this chapter.

The reason we distinguish between events and augmented events is that events are what

is usually of general interest. Events hide the state information that is not useful for anything but computing the next event. This is similar to hiding the state of the clocks when describing the state of a GSMP, and is commonly done with other formalisms.

5.2 Augmented State and Event

5.2.1 Augmented state

An augmented state includes the state information not just of the state variables, but of actions and reward variables as well. Let RM be a reward model. Recall IS is a set of initial values with P_S as a distribution function over IS. Thus, the initial state of the state variables is $\sigma_0 \in IS$ with probability $P_S(s_0)$.

Also recall that the initial state of the action state is given implicitly in 3.3.2. We call this AS_0. Finally, the initial state of the reward variable automata is given by ρ_0.

The *initial augmented state* of a Möbius model is given by $\varsigma_0 = (\sigma_0, AS_0, \rho_0)$ with probability $P_S(s_0)$. An *augmented state* is in general the triple $(\varsigma = (\sigma, AS, \rho))$.

5.2.2 Augmented event

The *augmented event* is the triple $\varepsilon = (\varsigma, \tau, a)$, where ς is an augmented state, τ is the sojourn time, and a is the action that completes. Note that we omit the resulting next state because it is uniquely defined by the augmented state, sojourn time, and action that completes. Furthermore, no other Möbius components use augmented events for computation, so it is unnecessary to include the next state for that reason.

Let the augmented state be ς_i. Initially, this is ς_0. In this section, we describe probabilistically what the event e_i will be, namely τ_i and a_i. Then, given e_i, what the next augmented state ς_{i+1} will be.

5.2.3 Computing τ_i and a_i

In order to describe the future behavior of the stochastic process, we must determine state sojourn time and the next action to complete. Let ς_i be the current augmented state. The evaluation process we describe takes an approach where by at any event in the process we discover a unique next action to complete and the time until it completes. If no unique action exists at one point in the process, further refinements are used to select a unique action.

The selection process proceeds briefly as follows. If only one action is enabled, that action

will complete. If several actions are enabled, the one that will complete first is the one with the smallest time to completion. If several actions have the same smallest time to completion, then the action with the highest rank completes. If several actions have the same highest rank, then weights are used among actions in the same action partition group. If actions of several partition groups are still competing, then the well-specified algorithm must be used to determine whether it matters which action partition group is selected to complete first (see Chapter 6). Once an action partition group is selected, weights are used to determine the probabilistic order among actions in that action partition group. The formal explanation follows.

Let $D(\varsigma_i) = \{d_a\}$ be the set of delay random variables described by the delay distribution function F_{d_a} for augmented state ς_i, which is described in Section 2.4.2. Note that $d_a \in D(\varsigma_i)$ iff $a.Enabled(\sigma_i) = true$ for $\sigma_i \in \varsigma_i$, that is, a delay random variable is in $D(\varsigma_i)$ iff the corresponding action is enabled. We frequently write only D if the augmented state ς_i is clear from context. From D, we can construct a set of "clocks," which is used to determine the sequence of action completions. Let $C(\varsigma_i)$ be the set of clocks in augmented state ς_i. (We frequently omit ς_i except for clarity, i.e., we simply write C.) The value of $c_a = d_a - (t_i - AS_i(a).\text{Start})$, that is, amount of time from t_i until the action completes. Let $A_C(\varsigma_i) = \{a : c_a \in C(\varsigma_i)\}$, or identically $A_C(\varsigma_i) = \{a : a.Enabled(\sigma_i) = true)\}$, the set of actions with active clocks. Again, we frequently omit ς_i and simply write A_C if the augmented state ς_i is clear from context. If $|A_C| = 1$, that is, only one action is enabled, then for $c \in C$ (we know $|C| = 1$), the sojourn time $\tau_i = c$, and the unique $a_i \in A_C$ completes. Thus, the augmented event is $\varepsilon_i = (\varsigma_i, \tau_i, a_i)$. Otherwise, if $|A_c| > 1$, more evaluation is necessary.

Let $C_{\min} = \min C$, the minimum completion time, and $A_{C_{\min}} = \{a : c_a = C_{\min}\}$, the set of actions with the earliest completion time. If $|A_{C_{\min}}| = 1$, then there is a unique action with the earliest completion time. This should be considered "normal" model behavior, the case that occurs most frequently in practice. That action is the unique $a_i \in A_{C_{\min}}$, and the sojourn time $\tau_i = C_{\min}$. Thus, the augmented event is $\varepsilon_i = (\varsigma_i, \tau_i, a_i)$. Otherwise, if $|A_{C_{\min}}| > 1$, more evaluation is necessary.

Let $r = \max\{a.Rank(\sigma_i) : a \in A_{C_{\min}}\}$, the highest rank among enabled actions with the same earliest completion time. Let $A^r_{C_{\min}} = \{a \in A : c_a \in C_{\min}, a.Rank(\sigma) = r\}$, the set of actions with the earliest completion time and the highest rank. If $|A^r_{C_{\min}}| = 1$, then there is a unique action among actions with the same earliest completion time with the highest rank. Let that action be $a_i \in A^r_{C_{\min}}$, and the sojourn time is $\tau_i = C_{\min}$. Thus, the augmented event is $\varepsilon_i = (\varsigma_i, \tau_i, a_i)$. Otherwise, if $|A^r_{C_{\min}}| > 1$, more evaluation is necessary.

If $|A^r_{C_{\min}}| > 1$, then we must determine whether the actions belong to separate action

partition groups. Recall AG is a partition of the actions, i.e., AG is a set of sets of actions. If $|\{G \in AG : G \cap A^r_{C_{\min}} \neq \emptyset\}| = 1$, i.e., actions in $A^r_{C_{\min}}$ belong to the same partition group, then the action that completes can be selected probabilistically. The probability of choosing action $a_i \in A^r_{C_{\min}}$ is given by

$$\Pr[a_i] = \frac{a_i.Weight(\sigma)}{\sum_{a' \in G_a \cap A^r_{C_{\min}}} a'.Weight(\sigma)} ,$$

where $G_a = G \in AG : a \in G$, the partition set containing a. If a_i is selected to complete, then the sojourn time is $\tau_i = C_{\min}$. Thus, the augmented event is $\varepsilon_i = (\varsigma_i, \tau_i, a_i)$.

If $|\{G \in AG : G \cap A^r_{C_{\min}}\}| > 1$, then the well-specified algorithm must be used. This is described in detain in Chapter 6. If the result of the well-specified checker is "pass," then any action partition group may be chosen arbitrarily. Once the partition group is chosen, an action from that partition group may be chosen probabilistically as described above.

The well-specified checker (described formally in the next chapter) will result in "pass" if the order does not "matter" in any measurable sense, and that the timed behavior of the stochastic process remains the same. We allow the behavior of the stochastic process to be ambiguous for zero time, as long as the ambiguity is not measurable by any of the reward variables. The well-specified algorithm checks whether this is the case. If, however, the well-specified check fails, then the ambiguity "matters" and the model is not sufficiently specified. The model cannot be solved due to these ambiguities. Solvers that detect this must abort the solution process.

If $C = \emptyset$, then the model execution halts.

Given ς_i, we have described how to compute τ_i and a_i to complete the augmented event $\varepsilon_i = (\varsigma_i, \tau, a_k)$. This algorithm is given in algorithmic form in Figure 5.1.

5.3 Computing ς_{i+1}

The previous section describes how to compute ε_i given ς_i. In this section, we describe how to compute ς_{i+1} given ε_i, which is a deterministic calculation.

5.3.1 Computing σ_{i+1}

First, we compute σ_{i+1}. This is straightforward. Let $\varepsilon_i = ((\sigma_i, AS_i, \rho_i), \tau_i, a_i)$. Then

$$\sigma_{i+1} = a_i.Complete(\sigma_i) .$$

If $\mathcal{C}(\varsigma_i) = \emptyset$ then **Halt**
Let $A_C(\varsigma_i) = \{a : c_a \in C(\varsigma_i)\}$
If $|A_C(\varsigma_i)| = 1$ then
    $a_i = a \in A_C(\varsigma_i)$
    Go to `selected`
End if
Let $C_{\min}(\varsigma_i) = \min C(\varsigma_i)$
Let $A_{C_{\min}}(\varsigma_i) = \{a : c_a = C_{\min}(\varsigma_i)\}$
If $|A_{C_{\min}}(\varsigma_i)| = 1$ then
    $a_i = a \in A_{C_{\min}}(\varsigma_i)$
    Go to `selected`
End if
Let $r = \max\{a.Rank(\sigma_i) : a \in A_{C_{\min}}(\varsigma_i)\}$
Let $A^r_{C_{\min}}(\varsigma_i) = \{a \in A_{C_{\min}}(\varsigma_i) : a.Rank(\sigma_i) = r\}$
if $|A^r_{C_{\min}}(\varsigma_i)| = 1$ then
    $a_i = a \in A^r_{C_{\min}}(\varsigma_i)$
    Go to `selected`
If $|\{G \in AG : G \cap A^r_{C_{\min}}(\varsigma_i)| > 1$ then
    Perform well-specified check
    If **Failed** then **Halt**
Let $G \in AG : G \cap A^r_{C_{\min}}(\varsigma_i) \neq \emptyset$ (uniquely or arbitrarily)
Choose $a_i \in G$ with probability

$$\Pr[a_i] = \frac{a_i.Weight(\sigma)}{\sum_{a' \in G \cap A^r_{C_{\min}}} a'.Weight(\sigma)},$$

`selected:` $\tau_i = C_{\min}$

Figure 5.1: Computing τ_i and a_i given ς_i.

5.3.2 Computing AS_{i+1}

Recall that

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

We compute $AS_{i+1}(a_j)$ based on a number of conditions. We present each of the conditions informally in the text below, and formally in algorithmic form in Figures 5.2 through 5.5.

Some of the conditions are simpler than others. The more complex conditions are more complex because they must take into account the various options available through the execution policy. Each of these complex cases is broken up into three subconditions, based on the three independent choices in the execution policy.

If $a_j.Enabled(\sigma_i) = false \wedge a_j.Enabled(\sigma_{i+1}) = false$ then
    // a_j remains disabled
    $AS_{i+1}(a_j) = AS_i(a_j)$
End if

If $a_j.Enabled(\sigma_i) = false \wedge a_j.Enabled(\sigma_{i+1}) = true$ then
    // a_j becomes enabled
    $AS_{i+1}(a_j)$.Start $= t_i$
    If $a_j.Policy(\sigma_{i+1}) = \circ \circ \mathrm{P} \wedge AS_i(a_j)$.Delay $\neq \emptyset$
    // preserve delay and effort functions
        $AS_{i+1}(a_j)$.Delay $= AS_i(a_j)$.Delay
        $AS_{i+1}(a_j)$.Effort $= AS_i(a_j)$.Effort
    Else
        $AS_{i+1}(a_j)$.Delay $= a_j.Delay(\sigma_{i+1})$
        $AS_{i+1}(a_j)$.Effort $= a_j Effort(\sigma_{i+1})$
    End if
    If $a_j.Policy(\sigma_{i+1}) = \mathrm{P} \circ \circ$
        // preserve worker effort
        $AS_{i+1}(a_j)$.WE $= AS_i(a_j)$.WE
    Else
        $AS_{i+1}(a_j)$.WE $= 0$
    End if
    If $a_j.Policy(\sigma_{i+1}) = \circ \mathrm{P} \circ$
        // preserve minimum task effort
        $AS_{i+1}(a_j)$.MTE $= AS_i(a_j)$.MTE
    Else
        $AS_{i+1}(a_j)$.MTE $= AS_{i+1}(a_j)$.WE
    End if
End if

Figure 5.2: Action state if action remains disabled or becomes enabled.

Remains disabled

If a_j remains disabled, that is, if $a_j.Enabled(\sigma_i) = a_j.Enabled(\sigma_{i+1}) = false$, then $AS_{i+1}(a_j) = AS_i(a_j)$, i.e., no change. This is illustrated in Figure 5.2.

Becomes enabled

If a_j becomes enabled, that is, if $a_j.Enabled(\sigma_i) = false$ and $a_j.Enabled(\sigma_{i+1}) = true$, then the following occurs. (This is shown in algorithmic form in Figure 5.2.) The start time is set to the current time. This is done through the expression $AS_{i+1}(a_j)$.Start $= t_i = \sum_{j=0}^{i} \tau_j$.

If $a_j.Enabled(\sigma_i) = true \wedge a_j.Enabled(\sigma_{i+1}) = true \wedge a_j \neq a_i \wedge$
$a_j.Interrupt(e_i) = false$
// a_j remains enabled, does not complete, is not interrupted
$AS_{i+1}(a_j) = AS_i(a_j)$
End if
If $a_j.Enabled(\sigma_i) = true \wedge a_j.Enabled(\sigma_{i+1}) = true \wedge a_j \neq a_i \wedge$
$a_j.Interrupt(e_i) = true$
// a_j remains enabled, does not complete, is interrupted
If $a_j.Policy(\sigma_{i+1}) = \circ \circ \mathrm{P}$
// preserve delay and effort functions
$AS_{i+1}(a_j).\mathrm{Delay} = AS_i(a_j).\mathrm{Delay}$
$AS_{i+1}(a_j).\mathrm{Effort} = AS_i(a_j).\mathrm{Effort}$
Else
$AS_{i+1}(a_j).\mathrm{Delay} = a_j.Delay(\sigma_{i+1})$
$AS_{i+1}(a_j).\mathrm{Effort} = a_j.Effort(\sigma_{i+1})$
End if
If $a_j.Policy(\sigma_{i+1}) = \mathrm{P} \circ \circ$
// preserve worker effort
Let $t_{jw} = AS_i(a_j).\mathrm{Effort}^{-1}(AS_i(a_j).\mathrm{WE}) + t_{i+1} - AS_i(a_j).\mathrm{Start}$
$AS_{i+1}(a_j).\mathrm{WE} = AS_{i+1}(a_j).\mathrm{Effort}(t_{jw})$
Else
$AS_{i+1}(a_j).\mathrm{WE} = 0$
End if
If $a_j.Policy(\sigma_{i+1}) = \circ \mathrm{P} \circ$
Let $t_{jw} = AS_i(a_j).\mathrm{Effort}^{-1}(AS_i(a_j).\mathrm{WE}) + t_{i+1} - AS_i(a_j).\mathrm{Start}$
Let $t_{jm} = AS_i(a_j).\mathrm{Effort}^{-1}(AS_i(a_j).\mathrm{MTE})$
Let $t_{je} = \max\{t_{jw}, t_{jm}\}$
$AS_{i+1}(a_j).\mathrm{MTE} = AS_{i+1}(a_j).\mathrm{Effort}(t_{je})$
Else
$AS_{i+1}(a_j).\mathrm{MTE} = AS_{i+1}(a_j).\mathrm{WE}$
End if
$AS_{i+1}(a_j).\mathrm{Start} = t_{i+1}$
End if

Figure 5.3: Action state if action remains enabled.

If $a_j.\mathit{Enabled}(\sigma_i) = \mathit{true} \wedge a_j.\mathit{Enabled}(\sigma_{i+1}) = \mathit{false} \wedge a_j \neq a_i$
    // a_j becomes disabled, does not complete
    If $a_j.\mathit{Policy}(\sigma_{i+1}) = \mathrm{P} \circ \circ$
        // preserve worker effort
        Let $t_{jw} = AS_i(a_j).\mathrm{Effort}^{-1}(AS_i(a_j).\mathrm{WE}) + t_{i+1} - AS_i(a_j).\mathrm{Start}$
        $AS_{i+1}(a_j).\mathrm{WE} = AS_i(a_j).\mathrm{Effort}(t_{jw})$
    Else
        $AS_{i+1}(a_j).\mathrm{WE} = 0$
    End if
    If $a_j.\mathit{Policy}(\sigma_{i+1}) = \circ \mathrm{P} \circ$
        // preserve minimum task effort
        Let $t_{jw} = AS_i(a_j).\mathrm{Effort}^{-1}(AS_i(a_j).\mathrm{WE}) + t_{i+1} - AS_i(a_j).\mathrm{Start}$
        Let $t_{jm} = AS_i(a_j).\mathrm{Effort}^{-1}(AS_i(a_j).\mathrm{MTE})$
        Let $t_{je} = \max\{t_{jw}, t_{jm}\}$
        $AS_{i+1}(a_j).\mathrm{MTE} = AS_i(a_j).\mathrm{Effort}(t_{je})$
    Else
        $AS_{i+1}(a_j).\mathrm{MTE} = AS_{i+1}(a_j).\mathrm{WE}$
    End if
    If $a_j.\mathit{Policy}(\sigma_{i+1}) = \circ \circ \mathrm{P}$
        // preserve delay and effort functions
        $AS_{i+1}(a_j).\mathrm{Delay} = AS_i(a_j).\mathrm{Delay}$
        $AS_{i+1}(a_j).\mathrm{Effort} = AS_i(a_j).\mathrm{Effort}$
    Else
        $AS_{i+1}(a_j).\mathrm{Delay} = \emptyset$
        $AS_{i+1}(a_j).\mathrm{Effort} = \emptyset$
    End if
    $AS_{i+1}(a_j).\mathrm{Start} = 0$
End if

Figure 5.4: Action state if action becomes disabled.

The rest of the action state is determined by the choice of execution policy. If the policy preserves the delay distribution and effort functions (indicated by $a_j.\mathit{Policy}(\sigma_{i+1}) = \circ\circ\mathrm{P}$), and if there is a delay distribution and effort function to preserve (indicated by $AS_i(a_j).\mathrm{Delay} \neq \emptyset$), then the new delay distribution and effort functions are the same as the previous delay distribution and effort functions. This is written formally as $AS_{i+1}(a_j).\mathrm{Delay} = AS_i(a_j).\mathrm{Delay}$ and $AS_{i+1}(a_j).\mathrm{Effort} = AS_i(a_j).\mathrm{Effort}$. If the delay distribution and effort functions are discarded, then new delay distribution and effort functions must be "sampled." This is done through the expressions $AS_{i+1}(a_j).\mathrm{Delay} = a_j.\mathit{Delay}(\sigma_{i+1})$ and $AS_{i+1}(a_j).\mathrm{Effort} = a_j.\mathit{Effort}(\sigma_{i+1})$.

If $a_j = a_i$
    // Action a_j completes
    If $a_j.Enabled(\sigma_{i+1}) = false$
        // a_j becomes disabled after completing
        $AS_{i+1}(a_j)$.Start $= 0$
        $AS_{i+1}(a_j)$.Delay $= \emptyset$
        $AS_{i+1}(a_j)$.Effort $= \emptyset$
        $AS_{i+1}(a_j)$.WE $= 0$
        $AS_{i+1}(a_j)$.MTE $= 0$
    Else
        // Action a_j completes and remains enabled
        $AS_{i+1}(a_j)$.Start $= t_{i+1}$
        $AS_{i+1}(a_j)$.Delay $= a_j.Delay(\sigma_{i+1})$
        $AS_{i+1}(a_j)$.Effort $= a_j.Effort(\sigma_{i+1})$
        $AS_{i+1}(a_j)$.WE $= 0$
        $AS_{i+1}(a_j)$.MTE $= 0$
    End if
End if

Figure 5.5: Action state if action completes.

Next, we consider the worker effort. If the worker effort is preserved, that is, if $a_j.Policy(\sigma_{i+1}) = \text{P} \circ \circ$, then the new worker effort is the same as the previous worker effort. This is expressed by $AS_{i+1}(a_j)$.WE $= AS_i(a_j)$.WE. Otherwise, if the worker effort is discarded, then the new worker effort is set to zero, i.e., $AS_{i+1}(a_j)$.WE $= 0$.

Finally, we consider the minimum task effort. If the minimum task effort is preserved, that is, if $a_j.Policy(\sigma_{i+1}) = \circ\text{P}\circ$, then the new minimum task effort is the same as the previous minimum task effort, i.e., $AS_{i+1}(a_j)$.MTE $= AS_i(a_j)$.MTE. Otherwise, if the minimum task effort is discarded, then the minimum task effort is set to be the new worker effort. This must be done to preserve the invariant WE $\geq$ MTE. This is discussed in more detail in Section 2.3.3.

Remains enabled

If a_j remains enabled and e_i is not an interrupting event, then the action state remains the same. This is written formally in Figure 5.3 simply as $AS_{i+1}(a_j) = AS_i(a_j)$.

Interrupted

Next, we consider the case where a_j remains enabled and e_i is an interrupting event for e_j. This case presumes that a_j is not the completing action. This condition is described formally in Figure 5.3. This condition holds if $a_j.\mathit{Enabled}(\sigma_i) = \mathit{true}$, $a_j.\mathit{Enabled}(\sigma_{i+1}) = \mathit{true}$, $a_j \neq a_i$, and $a_j.\mathit{Interrupt}(e_i) = \mathit{true}$. Recall that an interrupting event acts similar to a disabling event followed immediately by an enabling event.

As with several other conditions, what happens to the action state depends on the policy of the action. If the delay distribution and effort functions are preserved, then the new delay distribution functions and effort functions are the same as the previous ones. Formally, we write $AS_{i+1}(a_j).\text{Delay} = AS_i(a_j).\text{Delay}$ and $AS_{i+1}(a_j).\text{Effort} = AS_i(a_j).\text{Effort}$. Otherwise, if the delay distribution and effort functions are discarded, then they must be "sampled" from the current delay and distribution functions. I.e., $AS_{i+1}(a_j).\text{Delay} = a_j.\mathit{Delay}(\sigma_{i+1})$ and $AS_{i+1}(a_j).\text{Effort} = a_j.\mathit{Effort}(\sigma_{i+1})$.

Next, we consider the worker effort. If the worker effort is preserved, two steps are taken to determine what the new worker effort should be. First, we must compute the effective worker time. This is done by first computing the effective worker time contributed by any previous enablings and adding the amount of time since the last defining event. I.e., $t_{jw} = AS_i(a_j).\text{Effort}^{-1}(AS_i(a_j).\text{WE}) + t_{i+1} - AS_i(a_j).\text{Start}$. Next, this effective worker time is converted to worker effort using the new effort function, i.e., $AS_{i+1}(a_j).\text{WE} = AS_{i+1}(a_j).\text{Effort}(t_{jw})$. Otherwise, if the worker effort is discarded, then we simply set the new worker effort to zero.

Finally, we consider the minimum task effort. If the minimum task effort is preserved, then one of two things can occur. First, the worker effort up to this point can exceed the previous minimum task effort. (This may hold even if the worker effort is discarded.) In this case, the new minimum task effort must be set to the worker effort, or what the worker effort would be if it were preserved. The other, simpler condition is that the worker effort does not exceed the old minimum task effort. In this case, the new minimum task effort is preserved by converting the previous minimum task effort into an effective minimum task time, and then converting the effective minimum task time into the new minimum task effort using the (potentially) new effort function. The reason we perform those two conversions instead of simply copying the previous minimum task effort to the new minimum task effort is to mimic more closely the behavior of an action becoming disabled and then immediately becoming enabled.

Disabled

If a_j becomes disabled, then the following occurs. (This is shown in algorithmic form in Figure 5.4.) As with other cases, the new action state depends on the three policy conditions. However, since a disabled action may have no valid delay distribution or effort function, the policy conditions are checked in a different order and the equations are slightly different.

First, we check the worker effort condition. If the action preserves the worker effort, then we must first compute the effective worker time. This is the sum of the effective worker time contributed by the previous worker effort and the amount of time the action has been enabled since the last defining event. Formally, this is written as $t_{jw} = AS_i(a_j).\text{Effort}^{-1}($ $AS_i(a_j).\text{WE})+t_{i+1}-AS_i(a_j).\text{Start}$. Then, the effective worker time is converted to the worker effort using the *old* effort function (since there may not be a valid new effort function). I.e., $AS_{i+1}(a_j).\text{WE} = AS_i(a_j).\text{Effort}(t_{jw})$. Otherwise, if the worker effort is discarded, then the new worker effort is set to zero.

Next, if the minimum task effort is preserved, then one of two conditions can occur. First, the worker effort up to this point in time can exceed the previous minimum task effort. (This may be true even if the worker effort is discarded.) In this case, the new minimum task effort is set to what the worker effort would be if it were preserved. The other condition is that the worker effort does not exceed the minimum task effort. In this case, the new minimum task effort is the old minimum task effort, converted into the effective minimum task time, and then converted back to the minimum task effort. The two conversions are not strictly necessary; simply copying the previous minimum task effort to the new minimum task effort would be identical, but doing the computation this way makes the algorithm simpler.

Finally, if the delay distribution and effort functions are preserved, then the new delay distribution and effort functions are the same as the previous ones. Otherwise, if they are discard, the new delay distribution and effort functions are set to the empty set.

Completes

There are two sub-conditions if an action completes depending on whether the action is enabled in the new state. This condition is described formally in Figure 5.5. If a_j completes and is disabled in the new state, then the action state is set to the initial action state. I.e., $AS_{i+1}(a_j).\text{Start} = 0$, $AS_{i+1}(a_j).\text{Delay} = AS_{i+1}(a_j).\text{Effort} = \emptyset$, $AS_{i+1}(a_j).\text{WE} = 0$, and $AS_{i+1}(a_j).\text{MTE} = 0$.

Otherwise, if the action completes and is enabled in the new state, then the action state is set as follows. The start time is set to the current time, the delay distribution and effort functions are sampled, the worker effort is set to zero, and the minimum task effort is set to

zero.

5.3.3 Computing ρ_{i+1}

Computing $\rho_{i+1}(r)$ is straightforward: $\rho_{i+1}(r) = r.\delta(\rho_i(r), e_i)$.

5.4 Characterizations of Stochastic Process

We repeat taht we are considering two characterizations of the stochastic process. The first is the "stochastic process," which is a sequence of events. It is useful in defining reward variables, but it is not semi-Markov in that the future behavior is not defined as a function of the current event, but as a function of all past events. The augmented stochastic process is a sequence of augmented events. The state portion of the event is augmented to include information so that the stochastic process is semi-Markov: the future behavior is solely a function of the current event. However, in general, the augmented event is a continuous-state semi-Markov process, or a generalized semi-Markov process.

Since a *stochastic process* of a Möbius model represents a stochastic sequence of events $e_0, e_1, \ldots$, we can define it formally as the mapping

$$SP : \Omega \times \mathbb{N} \rightarrow E\,,$$

where Ω is a set of possible outcomes. Events can be expressed as a tuple of their constituent parts, and an appropriate Borel measurable subset defined on them.

The *augmented stochastic process* is the sequence of augmented events $\varepsilon_0, \varepsilon_1, \ldots$. Let $\mathcal{E}$ be the set of all augmented events of a Möbius model. Formally, then, the augmented stochastic process is the mapping

$$ASP : \Omega \times \mathbb{N} \rightarrow \mathcal{E}\,.$$

Again, Ω is a set of possible outcomes, and augmented events can be expressed as a tuple of their constituent parts, and an appropriate Borel measurable subset defined on them.

5.5 Conclusion

In this chapter, we formally defined the Möbius stochastic process and augmented stochastic process. The stochastic process is used for defining reward variables, and is also useful for

the informal description of the Möbius execution policy. To formalize the execution policy, we introduced an augmented state, augmented event, and augmented stochastic process. We presented an algorithm for computing the stochastic process inductively. The random nature of the stochastic process is contained in two parts of the algorithm, which are choosing the next action to complete, and the choice among multiple actions that have the same earliest completion time and the same rank, and belong to the same action partition group. The rest of the algorithm is deterministic.

A formal description of the execution policy is an integral part of the framework description. So far, we have described the structure of a model, including actions, state variables, and properties. We have described measures on the model using reward variables. Now, except for the well-specified check, which we discuss in the following chapter, we have described how the model executes. This is sufficient for defining and solving models within the Möbius framework.

CHAPTER 6

AN EFFICIENT WELL-SPECIFIED CHECKER

6.1 Introduction

Stochastic Petri nets have emerged as a popular way of expressing performance and dependability models. They have been successfully used in building performance and reliability models with complex structural interactions. The ability to generate a Markov chain in a fairly straightforward way makes analytical/numerical solution possible, and the relatively simple structure yields fast simulators when numerical solution is impractical.

Extensions to SPNs, including generalized SPNs (GSPNs) [57], allow for both timed and immediate transitions. A fundamental problem that arises from immediate transitions is that of what to do when multiple immediate transitions are enabled and hence may fire at the same time. The network may or may not behave differently depending on which transition is chosen to fire first. For an example, see Figure 6.1. The GSPN on the left has two immediate transitions competing, and the resulting state will differ based on which transition fires first. On the right, however, two immediate transitions compete, but the state will be the same regardless of which transition fires first. Thus, the ambiguity is acceptable unless it affects some reward variable's impulse rewards. A similar problem arises if several timed transitions have the same firing time, which can happen if the transition delay is deterministic or has a discrete or mixed distribution.

Historically, there have been several approaches to solving this problem. An early solution [57] was to assign probabilities to immediate transitions. Since it is not always easy or practical to foresee all the possible combinations of enabled immediate transitions, the state-space generator prompts the user to assign probabilities to the set of enabled immediate transitions. This solution could be tedious to the user for larger, complex models.

Figure 6.1: Unacceptable and acceptable ambiguity.

Later refinements to GSPNs assigned priorities and weights to immediate transitions [72]. The probability of choosing an immediate transition is then the ratio of the weight of the immediate transition to the sum of the weights of enabled immediate transitions. While this is a reasonable solution, it still requires that the user have some implicit knowledge of what sets of immediate transitions may be enabled at one time. This can be difficult for larger models. Also, it sometimes leads to over-specification, when the user is required to specify weights that are irrelevant (as on the right of Figure 6.1). Furthermore, the modeler may make an error in building the model. If the user is always required to provide arbitrary orders even when the order does not matter, and the user believes that the order does not matter but is wrong, the user will provide arbitrary orders and will subsequently get arbitrary results. Ideally, only orders that matter should be specified, and orders that do not matter should remain unspecified. A check should then be performed to determine whether the order really matters.

A different approach is to use an *extended conflict set* (ECS) [73]. Weights in transitions are used only in choosing among transitions within the ECS. This is an attempt to reduce the complexity of assigning weights to immediate transitions. The rule regarding ECSs is that the order in which immediate transitions of different ECSs fire may not matter; that means that the firing of a transition in one ECS may not affect the enabling or firing rule of a transition in a different ECS. Compliance with this rule is called *defined at the net level*. There are structural tests [73] to determine whether a conflict between ECSs exists, but these methods may declare that a conflict may exist where one does not exist. The only known accurate test is a comprehensive test at state-generation time.

A similar problem exists when more than one timed transition is scheduled to fire at the same time. That happens when the delay distribution is discrete or mixed. Priorities, weights, and ECSs may be applied, but unless there is a type of global weight-assignment system, or a test at state-generation time, there is a possibility of ambiguity.

There are several other tests that may be performed at state-generation time that are aimed at reducing the burden of specification and eliminating over-specification, while still ensuring that the behavior of a GSPN is completely specified. One of these tests is based on

the notion of a well-defined GSPN [74]. A GSPN is well-defined if the underlying stochastic process is completely described at every step, including all events that occur in zero time. While the state space is being explored, an efficient algorithm that does not add to the asymptotic running time of the state-generation algorithm can check whether the model is well-specified. However, the *well-defined* definition may be too restrictive, since some ambiguities do not change the behavior of the stochastic process in a measurable sense (for example, with respect to a reward structure). (For example, if the model on the right of Figure 6.1 has different impulse rewards for the top and bottom immediate transitions, then the ambiguity is measurable.) Therefore, a more relaxed definition, called *well-defined with respect to a reward structure*, was defined and applied [74]. We discuss some of the details of this definition later. An efficient algorithm to detect whether a GSPN is well-defined with respect to a reward structure has been given in [74].

In parallel with the evolution of GSPNs, a technique involving stochastic activity networks (SANs) [56], [75] has been developed to address a closely related problem. An activity (similar to a transition) has *cases*, which have probabilities assigned to them. When an activity completes, a case is chosen. Since SAN activities have cases, there is no need to use weights on instantaneous activities to specify choices between different behaviors in the SAN. However, if multiple instantaneous activities are enabled at the same time, there may be some ambiguity as to which activity completes first. As long as the probability of going to a next stable (similar to tangible) marking and obtaining an impulse reward is independent of the order in which instantaneous activities complete, we say that the SAN is "well-specified." A SAN must be well-specified for its behavior to be analyzed probabilistically. There are some subtle but important distinctions, which we discuss formally in Section 6.3, between "well-defined with respect to a reward structure" and "well-specified."

Algorithms have been developed to determine whether a SAN is well-specified [54], [76], [77]. However, those algorithms require the searching of all paths from a stable state to the set of next stable states. This makes the algorithm very computationally complex, which substantially slows down the state-space generation if there are many interacting instantaneous activities.

Möbius captures the possibly ambiguous behavior of GSPN extended conflict sets and SAN activities through the use of action partition groups. Möbius may accommodate a variety of ambiguity checks. Structural checks may be performed by model editors while the model is being constructed. However, checks that are performed at model-execution time must be performed by solvers. We would like to include both GSPN and SAN formalisms in the Möbius framework. This poses a unique problem. Since GSPNs and SANs have different criteria for allowing ambiguous behavior, solvers might need to have two checks for different

portions of a model.

We note that while the *well-defined* definition has historically been applied to GSPNs with ECSs and the *well-specified* definition has historically been applied to SANs, there is nothing unique about the definitions that ties them to these specific formalisms. GSPN ECSs are similar to SAN instantaneous activities, and GSPN immediate transitions are similar to SAN cases. (We discuss SANs in more detail in Section 6.2.1.) For simplicity, we discuss both definitions as applied to Möbius models.

In this chapter, we show that the two definitions (*well-specified* and *well-defined*) are in fact equivalent. We do this by first modifying the well-specified and *well-defined* definitions so that they apply to the Möbius stochastic process. This allows us to directly compare the two definitions on a common stochastic process. We then show by proof that the two definitions are equivalent. One important consequence of this is that we can use an efficient algorithm for performing the well-specified check. Another implication is that within the Möbius framework, we need only have one criterion, not two or more.

We begin, in Section 6.2, by reviewing SANs and introducing the configuration graph. In Section 6.3, we formally present the definitions of *well-specified* and *well-defined*. In Section 6.4, we prove that the two definitions are equivalent, and in Section 6.5 we show the algorithm as applied to the configuration graph. We conclude in Section 6.6.

6.2 SANs

6.2.1 Review of SANs

SANs are similar in some respects to stochastic Petri nets. A SAN is composed of places, activities (similar to transitions), input gates, and output gates. Places hold tokens, as they do in Petri nets. The number of tokens in places of the SAN is called the *marking* of the SAN. Formally, let P be the (finite) set of places in the SAN. A marking is then a mapping $\mu : P \to \mathbb{N}$.

Activities in a SAN may be either *timed* or *instantaneous*. Each timed activity has a (possibly general) probability distribution function assigned to it, which describes the time an activity takes to complete once it becomes enabled. Timed activities are drawn as hollow vertical bars. Instantaneous activities have no time distribution associated with them; the delay between enabling and completion is "instantaneous" with respect to time in the model. Instantaneous activities are drawn as solid vertical lines. Each activity has a (positive) number of cases associated with it, which probabilistically describe a choice of next markings a SAN may enter after the completion of the activity. A case is drawn as a

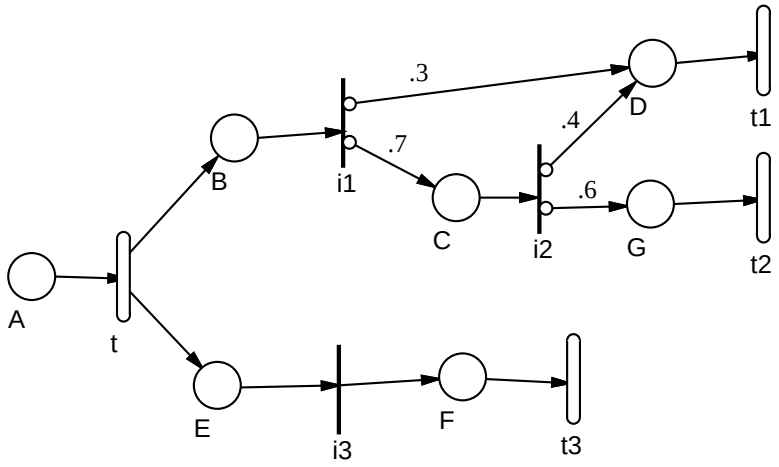

Figure 6.2: SAN example.

circle on an activity, usually on the right.

Each input gate has several input arcs and a single output arc. Input arcs are connected to places in the SAN, and the output arc connects to an activity. An input gate is made up of two functions: an input gate predicate, which is a general marking-dependent expression used to determine whether an activity is enabled, and an input gate function, which is a general marking-change function used to change the SAN marking when an activity completes. The input gate function may change the marking of any of the connected places. Input gates are drawn as right-pointing triangles.

An output gate has input arcs that connect activity cases to the output gate, and output arcs that connect the gate to places. An output gate also has a general function that describes how the marking changes when the connected activity completes; this function may change the marking of the connected places in an arbitrary way. Output gates are drawn as left-pointing triangles.

If a place is directly connected to an activity by an arc, the behavior of the SAN is the same as that of an SPN. When an activity completes, a case is chosen, the input gate functions are executed, and then the output gate functions of the output gates that are connected to the chosen case are executed.

An example SAN is given in Figure 6.2. This example shows a SAN with places, timed activities, and instantaneous activities with and without cases. Gates are a powerful extension, and allow for general, marking-dependent behavior; however, for the purpose of illustration, we omit gates from our example.

6.2.2 SAN execution

The execution of a SAN specifies how the marking of the SAN changes over time, and which activity completions and case selections cause these changes. An activity is *enabled* if all the places connected to the activity by input arcs have more than one token, as in Petri nets, and all the input gate predicates are true.

Instantaneous activities always have priority over timed activities; that is, if there are timed and instantaneous activities enabled in the same marking, an instantaneous activity will complete. If several timed activities are enabled and no instantaneous activities are enabled, the activity with the minimum delay is chosen to complete first.

The "yields" function specifies how the marking changes when an activity completes. We say that the completion of activity a and the choice of case c in marking μ *yields* marking μ', and we write $\mu \stackrel{a,c}{\rightarrow} \mu'$.

We make use of the reflexive, transitive closure of the yields relation, $\stackrel{*}{\rightarrow}$, to construct the set of reachable markings of a SAN. From the initial marking of a SAN, μ_0, we can define the set of all reachable markings in the SAN as $R(SAN, \mu_0) = \{\mu | \mu_0 \stackrel{*}{\rightarrow} \mu\}$. A reachable marking can be stable or unstable. A *stable marking* is one in which no instantaneous activities are enabled; an *unstable marking* is a marking in which one or more instantaneous activities are enabled. We denote the set of stable reachable markings as $SR(SAN, \mu_0) \subseteq R(SAN, \mu_0)$, and the set of unstable reachable markings as $UR(SAN, \mu_0) \subseteq R(SAN, \mu_0)$.

The execution of a SAN is described, formally, using configurations. A *configuration* is a triple $<\mu, a, c>$, and represents the completion of activity a and the choice of case c in marking μ. (Note the similarity to Möbius events.) We say that a configuration is *stable* if the marking of the configuration is a stable marking; a configuration is *unstable* if the marking of the configuration is unstable. We use the symbol "$-$" for "don't care" elements in configurations. The execution of a SAN is described in terms of a sequence of configurations called a *step*. There is a certain type of step called a *stable step*, which is a sequence of configurations in which the first configuration is stable, and subsequent configurations are unstable. The last configuration must yield a stable marking. (Note that the last configuration is not a stable configuration, but it must yield a stable marking.)

6.2.3 Configuration graph

We can use a configuration graph to help us describe the zero-timed behavior of a model, whether it be a SAN model, GSPN model, or Möbius model. To be general in our discussion, we apply it to a Möbius model. There are similarities between configuration graphs and state-space graphs of the unstable state between stable states. The primary difference is

that we distinguish between states in which there are nonquantified choices (e.g., zero-timed actions in different action partition groups) and those in which there are quantified choices. An expression is said to be *quantified* if the probability of the expression can be evaluated without further information; the expression is *nonquantified* if it is not quantified. These definitions help us considerably in our analysis.

A configuration graph (CG) is a rooted, connected, directed acyclic graph. Each node is identified with the state of a model, although the identification is not necessarily unique. A CG has a root node, which is the initial stable state, and leaves, which are stable states. The intermediate nodes correspond to unstable states. Thus, the CG is used to describe the evolution of a model from a stable state, through some some sequence of unstable states, to a set of next stable states. We label the nodes with states, e.g., σ_i. If a state is stable, then we use a hat, e.g., $\hat{\sigma}_i$.

Recall the notation we used in Section 5.2.3. Let $C_{\min}(\varsigma_i)$ be the earliest completion time in augmented state ς_i; let $A_{C_{\min}}(\varsigma_i) = \{a : c_a \in C_{\min}(\varsigma_i)\}$; and let $r = \max\{a.Rank(\sigma_i) : a \in A_{C_{\min}(\varsigma_i)}\}$ be the highest rank among enabled actions with the same earliest completion time. Then $A^r_{C_{\min}} = \{a \in A : c_a \in C_{\min}, a.Rank(\sigma) = r\}$, all as before. Note that state s_i is the state variable state of augmented state ς_i. Again, we omit ς_i and σ_i when the state is clear from context. Also recall from Section 3.3.3 that $AG : 2^{2^A}$ is a partition of the set of actions, so $\cup_{G \in AG} G = A$ and $\cap_{G \in AG} G = \emptyset$.

Recall the rule that if two actions have the same earliest completion time and the same highest rank, i.e., members of $A^r_{C_{\min}}$, then weights are used to determine probabilistically which completes first. However, if they belong to different action partition groups, the order in which partition groups are chosen remains unspecified.

Configuration graphs have two types of nodes: case nodes and decision nodes. A *decision node* is a node that corresponds to a state in which actions in $A^r_{C_{\min}}$ belong to different action partition groups. This node represents a state in which the next action to complete is ambiguous. Let $AG^r_{C_{\min}} = \{G \in AG : G \cap A^r_{C_{\min}} \neq \emptyset\}$, the set of partition groups that have competing actions. Let σ_i be the state associated with a decision node. We label the node σ_i. σ_i has a child node called σ_i^G (a "case node") for each $G \in AG^r_{C_{\min}}$; σ_i^G represents state σ_i, in which the action partition group G is chosen so that an action $a \in G$ (chosen probabilistically from $A^r_{C_{\min}} \cup G$) will complete. Arcs from σ_i to σ_i^G are not labeled, or are labeled with the partition group G.

The other type of node is a case node. A *case node* is a node that corresponds to a state in which actions in $A^r_{C_{\min}}$ belong to the same action partition group, or, if actions in $A^r_{C_{\min}}$ belong to different action partition groups, then one partition group $G \in AG^r_{C_{\min}}$ has been chosen. Case nodes correspond to states in which the next action to complete is not

ambiguous and is specified either deterministically or probabilistically. Let σ_i^G be a case node associated with state σ_i, where partition group G is chosen to complete first. Let σ_j be a state that is reached from state σ_i when action $a \in G$ completes. Using Möbius terminology, we can say that an event exists in the stochastic process $(\sigma_i, \circ, a, \sigma_j)$. The arcs from node σ_i to σ_j are labeled with the probability of going from σ_i to σ_j, and sometimes with the corresponding action also. Recall that the probability of going from σ_i to σ_j because of the completion of action a, written $\Pr[\sigma_i \rightarrow \sigma_j]$, or p_{ij}, is given by

$$\Pr[a] = \frac{a.Weight(\sigma_i)}{\sum_{a' \in G_a \cap A^r_{C_{\min}}} a'.Weight(\sigma_i)}.$$

The root node is a special node. We identify the root node with the stable state $\hat{\sigma}_\bullet$ and a particular completion time $c \in C_{\min}$, i.e., the earliest completion time. It is possible that several actions have the same earliest completion time, i.e., $|A^r_{C_{\min}}| > 1$. If that is the case, then the root node is a decision node. If only one action has the earliest completion time, i.e., $|A^r_{C_{\min}}| = 1$, then the root node is a case node with only one possible child node. We label the root node $\hat{\sigma}^c_\bullet$ throughout this chapter.

The leaf nodes of the graph are identified with the next stable states, and the intermediate nodes are identified with unstable, intermediate states. The children of the root node $\hat{\sigma}^c_\bullet$ are identified with the markings that are reachable from marking $\hat{\sigma}^c_\bullet$ by the completion of timed action $t \in A^r_{C_{\min}}$. Note that if $|A^r_{C_{\min}}| = 1$ and the child node of the root node is a stable marking, then it is not necessary to perform a check. In general, the well-specified check is only necessary when the configuration graph contains decision nodes.

To facilitate the following discussion, we introduce some new notation. The child set of a node σ_i is simply the set of child nodes in the configuration graph, and we write this as $Ch(\sigma_i)$. Let $EN(\hat{\sigma}^c_\bullet) = \{a \in A : a.Enabled(\hat{\sigma}^c_\bullet) = true, AS_i(a).\text{Delay}(0) = 0\}$, the set of nonzero timed actions enabled in state $\hat{\sigma}^c_\bullet$, i.e., the set of actions enabled in $\hat{\sigma}^c_\bullet$ that will not complete in zero time. Let $IEN(\sigma_i)$ be the set of zero-timed actions (actions that have nonzero probability of completing in zero time) enabled in the unstable marking σ_i. Formally, $IEN(\sigma_i) = \{a \in A : a.Enabled(\sigma) = true, AS_i(a).\text{Delay}(0) \neq 0\}$. If σ_i is a decision node, then $Ch(\sigma_i)$ contains case nodes of the form σ_i^G, $\forall G \in AG^r_{C_{\min}}$. If σ_i is a case node, then $Ch(\sigma_i)$ contains the nodes that can be reached from σ_i by the completion of some activity. Note that the existence of some case nodes may be conditioned upon the choice of an action partition group G, so that only actions from group G may complete. Note that sometimes we write σ_i as a label for a case node if we do not know whether it takes the form σ_i or s_i^G.

In our discussion, we restrict ourselves to stabilizing models. A model is *stabilizing* if the

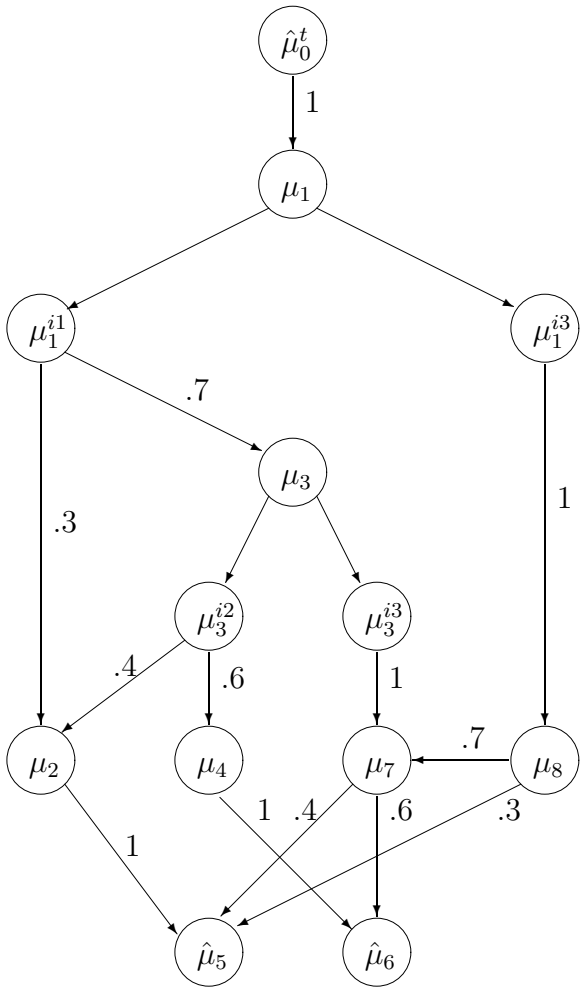

Figure 6.3: Configuration graph of a SAN.

number of steps (action completions) in any path from a stable state to some next stable state is finite. This means that the configuration graph must be acyclic.

Recall the example SAN in Figure 6.2. We show the configuration graph of this SAN in Figure 6.3, where place A initially has one token, and all other places have none. An enumeration of the markings of this configuration graph is given in Table 6.1.

6.3 Definitions

Before we discuss the definitions of *well-defined* and *well-specified*, we present a list of symbols and definitions for the reader's convenience. Note that many of these terms are defined with respect to a particular Möbius model.

σ_i A particular state of the state variables of a model. Subscripts on σ are used to distinguish

Table 6.1: Enumeration of markings.

Marking	A	B	C	D	E	F	G
μ_1	0	1	0	0	1	0	0
μ_2	0	0	0	1	1	0	0
μ_3	0	0	1	0	1	0	0
μ_4	0	0	0	0	1	0	1
$\hat{\mu}_5$	0	0	0	1	0	1	0
$\hat{\mu}_6$	0	0	0	0	0	1	1
μ_7	0	0	1	0	0	1	0
μ_8	0	1	0	0	0	1	0

states. State σ_i is not the same as state σ_j. However, state σ_i corresponds to the same state as state σ_i^G.

$\hat{\sigma}$ A stable state.

σ_0 The initial state of the model.

σ^G A state σ in which partition group G is chosen, so an action in G will complete first.

$\hat{\sigma}_\bullet^c$ A stable state σ_0 in which the earliest completion time is c.

ς_i An "augmented" state that includes all the "clock" values of the actions of a model.

$R(\sigma_0)$ Set of reachable states of a model.

$SR(\sigma_0)$ Set of reachable stable states of a model.

$UR(\sigma_0)$ Set of reachable unstable states of a model.

$EN(\sigma)$ The set of enabled timed actions in state σ.

$IEN(\sigma)$ The set of enabled zero-timed actions enabled in state σ.

$Ch(\sigma)$ The set of children of σ in the configuration graph. (The configuration graph is described above.)

$A^r_{C_{\min}}(\varsigma_i)$ Set of actions with the earliest completion time and highest rank in augmented state ς_i.

AG Action partition. It is a set of disjoint sets of actions that partition A, the set of actions.

$AG^r_{C_{\min}}(\varsigma_i)$ Set of action partition groups with elements in $A^r_{C_{\min}}(\varsigma_i)$. Formally, $AG^r_{C_{\min}}(\varsigma_i) = \{G \in AG : G \cap A^r_{C_{\min}}(\varsigma_i) \neq \emptyset\}$.

$\Pr[\hat{\sigma}^c_\bullet \rightsquigarrow \hat{\sigma}]$ The probability of reaching stable state $\hat{\sigma}$ from $\hat{\sigma}^c_\bullet$ by a stable step.

$\Pr[\hat{\sigma}^c_\bullet \rightsquigarrow \sigma_i]$ The probability of reaching unstable state σ_i from $\hat{\sigma}^c_\bullet$ by a sequence of zero-timed action completions.

$\Pr[\hat{\sigma}^c_\bullet \overset{\sigma_d}{\rightsquigarrow} \hat{\sigma}]$ The probability of reaching unstable state $\hat{\sigma}$ by a stable step that includes node σ_d.

$\Pr[\hat{\sigma}^c_\bullet \overset{\bar{\sigma}_d}{\rightsquigarrow} \sigma_i]$ The probability of reaching unstable state σ_i from $\hat{\sigma}^c_\bullet$ by a stable step that does not include node σ_d.

$\Pr[\hat{\sigma}^c_\bullet \rightsquigarrow \hat{\sigma}, im]$ The probability of reaching stable marking $\hat{\sigma}$ from $\hat{\sigma}^c_\bullet$ by a stable step and obtaining impulse reward im.

p_{ij} $\Pr[\sigma_i \to \sigma_j]$, where $\to$ implies $\exists a \in EN(\sigma_i) : \sigma_j = a_i.Complete(\sigma_j)$. Note that $p_{ij} > 0$ or it is undefined.

$NS(\hat{\sigma}^c_\bullet)$ The set of stable states reachable from $\hat{\sigma}^c_\bullet$ in a stable step.

$NU(\hat{\sigma}^c_\bullet)$ The set of unstable states reachable from state $\hat{\sigma}^c_\bullet$ in a stable step.

IM The set of allowable impulse rewards, which is some countable subset of $\mathbb{R}$.

Quantified A probability is *quantified* if it can be computed without any additional information. A probability is said to be *nonquantified* if it is not quantified.

Stable event The event $(\sigma, \tau, a, \sigma')$ is *stable* if σ is a stable state, i.e., $\tau > 0$. The event is *unstable* if σ is an unstable state, i.e., $\tau = 0$.

Stable step A sequence of events that consists of the following: first, a stable event; then, a number of unstable events; and finally, a terminating event that yields a stable event. The sequence must follow a legal execution of a Möbius model.

Stabilizing model A model is *stabilizing* if the number of events in a stable step from any stable state is finite.

6.3.1 Well-defined

We begin by describing the approach taken in [74], where the term *well-defined* is introduced. There are actually three different definitions (we believe) of *well-defined* in [74]. The underlying stochastic process of an SPN model is defined to include the state of the SPN, the time during which state changes occur, and the sequence of transition firings. SANs use the sequence of configurations to describe the underlying stochastic process, and Möbius uses events. Although the descriptions of the underlying stochastic process differ between SANs, GSPNs, and Möbius, all afford enough information that it is possible to apply the definition of *well-specified* or *well-defined*.

A *well-defined* GSPN is informally defined as a GSPN in which the underlying stochastic process is fully determined. Nondeterminism may occur when several enabled immediate transitions belong to different extended conflict sets (ECSs). The order in which transitions fire may affect the probability of reaching some tangible state, and the order is not specified. According to the definition of *well-defined*, sequences of transition firings, even if they happen in zero time, must always yield a completely determined underlying stochastic process. This allows for no ambiguity, which means that if more than one immediate transition is enabled, all of them must belong to the same ECS. If applied to SANs, that would mean that only one instantaneous activity could be enabled at any one time. As applied to Möbius, it would mean that all actions in the set $A^r_{C_{\min}}$ must belong to the same action partition group. This has the potential of being overly restrictive.

A more relaxed definition provided by Ciardo and Zijal [74] is that of *well-defined with respect to a reward structure*. Simply stated, this definition requires that the reward structure be fully determined, or quantified. This has some interesting side effects. For example, any SPN with no (or trivial) reward structure is well-defined with respect to that reward structure.

A more practical definition of *well-specified* is also offered in [74], in algorithm form. It is a more restrictive form of the definition of *well-defined with respect to a reward structure*. It adds two additional restrictions. First, the probability of reaching a stable state from any unstable state must be quantified (no probability confusion). Second, the probability of obtaining an impulse reward by reaching a stable state from any unstable state must be quantified (no reward confusion).

To state this precisely, we introduce some notation. Let $NU(\sigma_i)$ be the set of unstable states reachable from σ_i in zero time; let $NS(\hat{\sigma}^c_\bullet)$ be the set of stable states reachable by a stable step from marking $\hat{\sigma}_\bullet$ given completion time c; and let IM be the set of all impulse rewards. Note that for SANs and GSPNs, the root node must be a case node, because all

timed transitions and activities are required to have continuous distributions. This restriction is relaxed in Möbius, so the root node may be a decision node.

Formally, a Möbius model holds to the more restrictive definition of *well-defined with respect to a reward condition* if $\forall \hat{\sigma}_\bullet \in SR(\sigma_0)$, for $c \in \mathcal{C}_{\min}$, $\forall \sigma_i \in NU(\hat{\sigma}^c_\bullet)$, and $\forall \hat{\sigma} \in NS(\hat{\sigma}^c_\bullet)$, then the following are true:

1. $\Pr[\sigma_i \rightsquigarrow \hat{\sigma}]$ is quantified (no probability confusion).

2. $\Pr[\sigma_i \rightsquigarrow \hat{\sigma}, im]$ is quantified $\forall im \in IM$ (no reward confusion).

This is the (formal) definition of *well-defined* that we are adopting and that we will hereafter use in this thesis.

The algorithm for checking whether a model is well-defined involves a depth-first traverse of the configuration graph, and a determination of the probability of reaching a leaf node recursively. At each decision node, the probability of reaching each leaf node must be independent of which path is chosen from that decision node, i.e., the distribution must be equal for all child nodes. If computing and comparing the distribution at each node in the configuration graph can be done in constant time, and $G = (E, V)$ is the configuration graph, then the algorithm runs in $O(E + V)$ time.

6.3.2 Well-specified

The definition of *well-specified* essentially says that the marking of the SAN and the reward variables must be completely described at all points in time. Nonquantified choices (i.e., multiple enabled instantaneous activities) that introduce some ambiguity are allowable so long as the distribution of next stable markings and accumulated reward is independent of these nonquantified choices. The definition of *well-specified* requires that from a stable marking, the distribution of next stable markings must be quantified. However, it does not appear to require that the distribution of next stable markings be quantified for all unstable markings, as the definition of well-defined requires. For that reason, previous attempts to design an algorithm for the well-specified check have enumerated all possible paths in the configuration graph. We show in the next section that this apparent ambiguity cannot exist.

A Möbius model is well-specified if for any stable marking, the probability of reaching a next stable marking and obtaining an impulse reward is quantified. We say a model is *well-specified* if $\forall \hat{\sigma}_\bullet \in SR(\sigma_0)$, for $c \in C_{\min}$, and $\forall \hat{\sigma} \in NS(\hat{\sigma}^c_\bullet)$, then the following are true:

1. $\Pr[\hat{\sigma}^c_\bullet \rightsquigarrow \hat{\sigma}]$ is quantified, and

2. $\Pr[\hat{\sigma}^c_\bullet \rightsquigarrow \hat{\sigma}, im]$ is quantified $\forall im \in IM$.

There is a subtle but important difference between the definitions of *well-specified* and *well-defined*. Let $\hat{\sigma}^c_\bullet$ be the root of a configuration graph. The definition of *well-specified* requires that the probability of reaching the next stable marking and obtaining an impulse reward be quantified from $\hat{\sigma}^c_\bullet$, the root of the configuration graph. The definition of *well-defined* requires that the probability of reaching the next stable marking and obtaining an impulse reward be quantified for *all* nodes in the configuration graph.

This distinction is important, because the algorithms derived from these two definitions are substantially different in their computational complexity. A well-specified model, by definition, is assumed to be capable of having some unstable state for which the probability of reaching the next stable state and obtaining an impulse reward is not quantified. This causes the well-specified algorithm to be computationally complex, because it must enumerate all possible paths.

The algorithm to check whether a model is well-specified, which was developed in [54], involves enumerating and storing all possible paths (stable steps); this can be prohibitive for models with larger networks of instantaneous activities, or even for models with smaller networks that are frequently used. If $G = (E, V)$ is the configuration graph, then the algorithm executes in $O(2^E)$ time.

In the next section, we show that the definitions of *well-specified* and *well-defined* are actually equivalent. Hence, the algorithm to test whether a model is well-defined may be used to determine whether a model is well-specified.

6.4 Equivalence of Definitions

6.4.1 Condition 1

Recall that a model is well-specified if $\forall \hat{\sigma}_\bullet \in SR(\sigma_0)$, for $c \in C_{\min}$, and $\forall \hat{\sigma} \in NS(\hat{\sigma}^c_\bullet)$, the following are true:

1. $\Pr[\hat{\sigma}^c_\bullet \rightsquigarrow \hat{\sigma}]$ is quantified, and
2. $\Pr[\hat{\sigma}^c_\bullet \rightsquigarrow \hat{\sigma}, im]$ is quantified $\forall im \in IM$.

We begin by considering the first part of the definition, and in Section 6.4.2 we consider the second part.

We begin by stating three rules that are evident and easily derived from the definition of a model. We write all probabilities with respect to some configuration graph rooted at $\hat{\sigma}^c_\bullet$. Thus, we write $\Pr[\sigma_i \rightsquigarrow \sigma_j]$ and implicitly mean that the expression is defined with respect

to a configuration graph rooted at $\hat{\sigma}_\bullet^c$. Let $\hat{\sigma}_\bullet \in SR(\sigma_0)$, $c \in C_{\min}$, and $\hat{\sigma}, \hat{\sigma}' \in NS(\hat{\sigma}_\bullet^c)$, $\hat{\sigma} \neq \hat{\sigma}'$.

Rule 1. If node σ_i is a case node, then $\Pr[\sigma_i \rightsquigarrow \hat{\sigma}] = \sum_{\sigma_j \in Ch(\sigma_i)} \Pr[\sigma_j \rightsquigarrow \hat{\sigma}] p_{ij}$.

Rule 2. $\Pr[\hat{\sigma} \rightsquigarrow \hat{\sigma}] = 1$.

Rule 3. $\Pr[\hat{\sigma} \rightsquigarrow \hat{\sigma}'] = 0$.

Note that p_{ij} is always greater than zero.

Now we state our first condition.

Condition 1 *If node σ_i is a decision node, and $\hat{\sigma} \in NS(\hat{\sigma}_\bullet^c)$, then* $\Pr[\sigma_i^G \rightsquigarrow \hat{\sigma}] = \Pr[\sigma_i^H \rightsquigarrow \hat{\sigma}]$, $\forall G, H \in AG_{C_{\min}}^r$.

This leads us to our first theorem.

Theorem 6.1 *Condition 1 is a necessary and sufficient condition for a model to be well-specified with respect to the first part of the definition of* well-specified.

Proof: If Condition 1 holds, then $\Pr[\sigma_i \rightsquigarrow \hat{\sigma}]$ is quantified for all nodes σ_i in the configuration graph, so it is certainly quantified for the root node. This shows that Condition 1 is sufficient.

To show that Condition 1 is a necessary condition, we assume, for contradiction, that a well-specified model exists that does not meet Condition 1.

First, we note that the definition of *well-specified* implies that $\Pr[\hat{\sigma}_\bullet^c \rightsquigarrow \hat{\sigma}]$ is quantified for all $\hat{\sigma} \in NS(\hat{\sigma}_\bullet^c)$. Condition 1 implies that $\Pr[\sigma_i \rightsquigarrow \hat{\sigma}]$ is quantified for all σ_i in the configuration graph. For a contradiction, we can assume that in some configuration graph, Condition 1 does not hold for some nodes.

Based on the contradictory assumption, we can infer that there exists some node, call it σ_i, where $\Pr[\sigma_i \rightsquigarrow \hat{\sigma}]$ is quantified for some arbitrary $\hat{\sigma} \in NS(\hat{\sigma}_\bullet^c)$, but is not quantified for some children of σ_i. (If no node σ_i exists, then our contradictory assumption does not hold.)

There are two possibilities: σ_i is either a decision node or a case node. If σ_i is a decision node, then $\Pr[\sigma_i^G \rightsquigarrow \hat{\sigma}]$ is nonquantified for some $G \in AG_{C_{\min}}^r$ and for some $\hat{\sigma} \in NS(\hat{\sigma}_\bullet^c)$; therefore, $\Pr[\sigma_i \rightsquigarrow \hat{\sigma}]$ is nonquantified because it depends upon which action partition group is chosen. This violates our assumption, so σ_i is not a decision node.

Thus, σ_i must be a case node. Therefore, we can write

$$\Pr[\sigma_i \rightsquigarrow \hat{\sigma}] = \sum_{\sigma_j \in Ch(\sigma_i)} \Pr[\sigma_j \rightsquigarrow \hat{\sigma}] p_{ij} \,.$$

Our assumption says that for some σ_j, $\Pr[\sigma_j \rightsquigarrow \hat{\sigma}]$ is not quantified; that is, it is dependent on choices made at (successor) decision nodes.

Before we proceed, we introduce a new term. A *decision* specifies which action partition group is chosen in some decision node. We write a decision for group G at node σ_i as d_i^G. The decision d_i^G thus quantifies all nonquantified behavior in marking σ_i, and in particular, which partition group is chosen. A decision imposes an order where none is specified by the model.

A *decision vector* D is a vector of decisions in which each decision node in the configuration graph has a decision associated with that node in D. A decision vector prescribes all non-probabilistic choices in the configuration graph, and hence quantifies $\Pr[\circ \rightsquigarrow \hat{\sigma}|D]$ in general, and $\Pr[\sigma_j \rightsquigarrow \hat{\sigma}|D]$, $\forall \sigma_j \in NU(\hat{\sigma}_\bullet^c)$ in particular.

We can then define $\mathcal{D}$ to be a subspace defined by a set of decision vectors such that $\Pr[\sigma_j \rightsquigarrow \hat{\sigma}|D]$ is the same for all $D \in \mathcal{D}$. We claim that there must exist a decision vector $D' \notin \mathcal{D}$ such that D and D' differ by only one decision. To see this, first realize that the decision space is a regular, n-dimensional, finite, discrete space, where n is the number of decision nodes. There exists a finite path through the space that visits every point in the space, and moves in only one dimension at a time. If the path starts in $\mathcal{D}$, then either the path never leaves $\mathcal{D}$ and $\mathcal{D}$ is the entire space (a contradiction), or at some point the path leaves $\mathcal{D}$. Consider the two adjacent nodes along the path, one in $\mathcal{D}$, and one outside of $\mathcal{D}$. Since the path moves in only one dimension, the two nodes differ by only one decision.

Let the decision in which D and D' differ be called d_d^G and d_d^H; that is, D is identical to D' except that d_d^G is an element in D and d_d^H is an element in D'. This decision occurs at node σ_d.

Now we have two decision vectors that differ by only one decision and yield different quantified probabilities. Recall that we assume that $\Pr[\sigma_i \rightsquigarrow \hat{\sigma}]$ is quantified, but for some child nodes $\sigma_j \in Ch(\sigma_i)$, $\Pr[\sigma_j \rightsquigarrow \hat{\sigma}]$ is not quantified. We can then write

$$\Pr[\sigma_i \rightsquigarrow \hat{\sigma}] = \sum_{\sigma_j \in Ch(\sigma_i)} \Pr[\sigma_j \rightsquigarrow \hat{\sigma}] p_{ij}$$

as two separate equations:

$$\Pr[\sigma_i \rightsquigarrow \hat{\sigma}] = \sum_{\sigma_j \in Ch(\sigma_i)} \Pr[\sigma_j \rightsquigarrow \hat{\sigma}|D] p_{ij}\,, \tag{6.1}$$

$$\Pr[\sigma_i \rightsquigarrow \hat{\sigma}] = \sum_{\sigma_j \in Ch(\sigma_i)} \Pr[\sigma_j \rightsquigarrow \hat{\sigma}|D'] p_{ij}\,. \tag{6.2}$$

Taking the difference between (6.1) and (6.2), we get

$$0 = \sum_{\sigma_j \in Ch(\sigma_i)} \left(\Pr[\sigma_j \rightsquigarrow \hat{\sigma}|D] - \Pr[\sigma_j \rightsquigarrow \hat{\sigma}|D']\right) p_{ij} \,. \tag{6.3}$$

Now for a chosen $\sigma_j \in Ch(\sigma_i)$, we can partition the set of paths $\sigma_j \rightsquigarrow \hat{\sigma}$ into paths that include σ_d and paths that do not include σ_d. Recall that σ_d is the node at which the two decision vectors (D and D') differ in their decision. We can write

$$\begin{aligned} \Pr[\sigma_j \rightsquigarrow \hat{\sigma}|D] &= \Pr[\sigma_j \overset{\sigma_d}{\rightsquigarrow} \hat{\sigma}|D] + \Pr[\sigma_j \overset{\bar{\sigma}_d}{\rightsquigarrow} \hat{\sigma}|D] & (6.4)\\ \Pr[\sigma_j \rightsquigarrow \hat{\sigma}|D'] &= \Pr[\sigma_j \overset{\sigma_d}{\rightsquigarrow} \hat{\sigma}|D'] + \Pr[\sigma_j \overset{\bar{\sigma}_d}{\rightsquigarrow} \hat{\sigma}|D'] & (6.5) \end{aligned}$$

We make use of the following equalities.

$$\begin{aligned} \Pr[\sigma_j \overset{\sigma_d}{\rightsquigarrow} \hat{\sigma}|D] &= \Pr[\sigma_j \rightsquigarrow \sigma_d|D] \Pr[\sigma_d^G \rightsquigarrow \hat{\sigma}|D] \\ \Pr[\sigma_j \overset{\sigma_d}{\rightsquigarrow} \hat{\sigma}|D'] &= \Pr[\sigma_j \rightsquigarrow \sigma_d|D'] \Pr[\sigma_d^H \rightsquigarrow \hat{\sigma}|D'] \\ \Pr[\sigma_j \overset{\bar{\sigma}_d}{\rightsquigarrow} \hat{\sigma}|D] &= \Pr[\sigma_j \overset{\bar{\sigma}_d}{\rightsquigarrow} \hat{\sigma}|D'] \\ \Pr[\sigma_j \rightsquigarrow \sigma_d|D] &= \Pr[\sigma_j \rightsquigarrow \sigma_d|D'] \end{aligned}$$

Finally, the difference between (6.4) and (6.5) becomes

$$\begin{aligned} &\Pr[\sigma_j \rightsquigarrow \hat{\sigma}|D] - \Pr[\sigma_j \rightsquigarrow \hat{\sigma}|D'] = \\ &\quad \Pr[\sigma_j \rightsquigarrow \sigma_d|D] \left(\Pr[\sigma_d^G \rightsquigarrow \hat{\sigma}|D] - \Pr[\sigma_d^H \rightsquigarrow \hat{\sigma}|D']\right) \,. \end{aligned}$$

Now we can restate (6.3) and substitute:

$$\begin{aligned} 0 &= \sum_{\sigma_j \in Ch(\sigma_i)} \left(\Pr[\sigma_j \rightsquigarrow \hat{\sigma}|D] - \Pr[\sigma_j \rightsquigarrow \hat{\sigma}|D']\right) p_{ij} \\ &= \sum_{\sigma_j \in Ch(\sigma_i)} \Pr[\sigma_j \rightsquigarrow \sigma_d|D] \left(\Pr[\sigma_d^G \rightsquigarrow \hat{\sigma}|D] - \Pr[\sigma_d^H \rightsquigarrow \hat{\sigma}|D']\right) p_{ij} \\ &= \left(\Pr[\sigma_d^G \rightsquigarrow \hat{\sigma}|D] - \Pr[\sigma_d^H \rightsquigarrow \hat{\sigma}|D']\right) \sum_{\sigma_j \in Ch(\sigma_i)} \Pr[\sigma_j \rightsquigarrow \sigma_d|D] p_{ij} \end{aligned}$$

Now we have the equation in a form so that we can easly reason about them. The equation is in the form of a product of two terms, and the product is zero, so at least one of the terms must be zero. However, by definition, $p_{ij} > 0$. For some $\sigma_j \in C(\sigma_i)$, $\Pr[\sigma_j \rightsquigarrow \sigma_d, D] > 0$,

because otherwise, the choice at σ_d would not affect $\Pr[\sigma_j \rightsquigarrow \hat{\sigma}]$. Therefore, the summation must sum to a value greater than zero.

Now we examine the other term. If $\Pr[\sigma_d^G \rightsquigarrow \hat{\sigma}|D] = \Pr[\sigma_d^H \rightsquigarrow \hat{\sigma}|D'], \forall G, H \in AG^r_{C_{\min}}$, then $\Pr[\sigma_d \rightsquigarrow \hat{\sigma}]$ would be quantified. By assumption, we reasoned that there must exist a node that was not quantified, and called it σ_d. Therefore, $\Pr[\sigma_d^G \rightsquigarrow \hat{\sigma}|D] - \Pr[\sigma_d^H \rightsquigarrow \hat{\sigma}|D']$ must be nonzero.

Both terms cannot be nonzero and have a product that is zero. This is a contradiction. Therefore, σ_i may not be a decision node. Since σ_i is neither a case node nor a decision node, it must not exist. Thus, Condition 1 is both necessary and sufficient for a model to be well-specified with respect to the first part of the definition. ■

Restated, the first part of the definition of *well-specified* says that $\Pr[\hat{\sigma}_\bullet^c \rightsquigarrow \hat{\sigma}]$ must be quantified for all $\hat{\sigma}_\bullet^c \in SR(\sigma_0)$. Condition 1 implies that a model is well-specified if and only if $\Pr[\hat{\sigma}_\bullet^c \overset{\sigma_i}{\rightsquigarrow} \hat{\sigma}], \forall \sigma_i \in NU(\hat{\sigma}_\bullet^c)$, is quantified. Thus, there can be no unstable state in which the probability of reaching some next stable state in a stable step is not quantified. Notice that this is also the first part of the definition of *well-defined*.

Specifically, this result provides us with a condition that can be checked at each node in the configuration graph. Condition 1 holds for all nodes in the configuration graph, for all configuration graphs generated by a model, if and only if the model is well-specified with respect to the first definition of *well-specified*. Next, we address the second part of the definition of *well-specified*.

6.4.2 Condition 2

The second part of the definition of *well-specified* is that the probability of obtaining a certain impulse reward when entering a particular next stable marking, given that the earliest completion time in $\hat{\sigma}_\bullet$ is c, is independent of nonquantified activity choices.

We define $\Pr[\hat{\sigma}_\bullet^c \rightsquigarrow \hat{\sigma}, im]$ to be the probability that the next stable marking from $\hat{\sigma}_\bullet^c$ is $\hat{\sigma}$ and the impulse reward is im. Let IM be the set of all possible impulse reward values. Note that then $\Pr[\hat{\sigma}_\bullet^c \rightsquigarrow \hat{\sigma}] = \sum_{im \in IM} \Pr[\hat{\sigma} \rightsquigarrow \sigma, im]$. Again, for convenience we assume that all probabilities are in reference to a configuration graph rooted by $\hat{\sigma}_\bullet^c$, so we write $\Pr[\sigma_i \rightsquigarrow \hat{\sigma}]$ and implicitly mean that the expression is written with respect to a configuration graph rooted at $\hat{\sigma}_\bullet^c$.

From the definition of a Möbius model, we can derive the following rules. Let $\hat{\sigma}, \hat{\sigma}' \in NS(\hat{\sigma}_\bullet^c)$, $\hat{\sigma} \neq \hat{\sigma}'$.

Rule 1. Let σ_i^G be a case node, and recall that $\mathcal{C}(\sigma_i, 0, a, \sigma_j)$ is the impulse reward for a

completing in state σ_i resulting in state σ_j in 0 time. Then

$$\Pr[\sigma_i^G \rightsquigarrow \hat{\sigma}, im] = \sum_{\substack{a \in IEN(\sigma_i) \cup G_a, \\ \sigma_j = a.Complete(\sigma_i)}} \Pr[\sigma_j \rightsquigarrow \hat{\sigma}, im - \mathcal{C}(\sigma_i, 0, a, \sigma_j)] p_{ij}$$

Rule 2.

$$\Pr[\hat{\sigma} \rightsquigarrow \hat{\sigma}, im] = \begin{cases} 1 & im = 0\,, \\ 0 & im \neq 0\,. \end{cases}$$

Rule 3. $\Pr[\hat{\sigma} \rightsquigarrow \hat{\sigma}', im] = 0$.

Note that this proof allows for the most general, time- and state-dependent impulse rewards.

To address the second part of the definition of *well-specified*, we introduce a second condition.

Condition 2 *If node σ_i is a decision node, then* $\Pr[\sigma_i^G \rightsquigarrow \hat{\sigma}, im] = \Pr[\sigma_i^H \rightsquigarrow \hat{\sigma}, im]$, $\forall G, H \in AG^r_{C_{\min}}$.

This leads us to the second theorem.

Theorem 6.2 *Condition 2 is a necessary and sufficient condition for a model to be well-specified with respect to the second part of the definition of* well-specified.

Proof: The proof follows the same form as the proof for Theorem 6.1. ■

As we observed above, $\Pr[\hat{\sigma}_\bullet^c \rightsquigarrow \hat{\sigma}] = \sum_{im \in IM} \Pr[\hat{\sigma}_\bullet^c \rightsquigarrow \hat{\sigma}, im]$. Notice that the impulse rewards strictly partition the event $\{\hat{\sigma}_\bullet^c \rightsquigarrow \hat{\sigma}\}$. From this, we can deduce the following corollary.

Corollary 6.1 *Condition 2 implies Condition 1.*

The implication of this is, naturally, that only Condition 2 needs to be checked.

Notice that the second condition implies that $\Pr[\sigma_i \rightsquigarrow \hat{\sigma}, im]$ is quantified (no reward confusion). This is exactly the same as the second part of the definition of *well-defined*. Thus, we can state another corollary.

Corollary 6.2 *A model is well-specified if and only if it is well-defined.*

The definition of *well-specified* seems to be broader than the definition of *well-defined*, and one might presume, as some have ([54], [74]), that some model may exist that would meet the well-specified condition but not meet the well-defined condition. We have shown that this is *not* the case, and that the two definitions are in fact equivalent.

The direct consequence of this discovery is that the algorithm to perform the well-specified check can safely be replaced with the the algorithm for the well-defined check. The well-defined algorithm also unwittingly performs the well-specified check.

Note that the equivalence of the definitions depends on a subtle detail of the way we defined the configuration graph. We say that an arc exists from case node σ_i to σ_j if there is a finite probability of going from σ_i to σ_j. If the probability is 0, no arc exists. In Möbius, it is not possible for an action to have 0 weight, so that is not an issue. However, in a formalism like SANs, it may be possible to define a case that has probability 0. Technically, the stochastic process is defined so that a sample path exists that goes from σ_i to σ_j, but the probability of taking such a sample path is 0. Thus, it is possible that $\Pr[\hat{\sigma}_i \rightsquigarrow \hat{\sigma}]$ is not quantified and the model is well-specified. However, this could only be true if $\Pr[\hat{\sigma}^c_\bullet \rightsquigarrow \hat{\sigma}_i] = 0$. Thus, we can precisely say that the sample paths of a model execution that satisfies the well-specified condition but not the well-defined condition will occur with probability zero.

6.5 Well-Specified Checker

6.5.1 Algorithm

Here we present a new algorithm for doing the well-specified check. Naturally, it is similar to the algorithm for the well-defined check [74]. The algorithm performs a depth-first search, applying the rules and the condition at each node in the configuration graph.

To describe the distribution of $NS(\hat{\sigma}^c_\bullet)$, we use a vector P that is uniquely indexed by two elements: a marking and an impulse reward. Thus, the vector entry $P(\hat{\sigma}, im)$ at a node σ_i is $Pr[\sigma_i \rightsquigarrow \hat{\sigma}, im]$.

In one instance, we use a vector shorthand notation $P(\sigma_i, \circ) = P(\sigma_i, \circ) + P'(\sigma_j, \circ - im)p_{ij}$. This involves two constructs. First, there is the vector-scalar multiplication $P(\sigma, \circ) = P'(\sigma_i, \circ)p_{ij}$, which means simply $\forall m \in IM,\ P(\sigma, m) = P(\sigma_j, m)p_{ij}$. The second construct is $P(\sigma, \circ) = P'(\sigma_i, \circ) + P''(\sigma_j, \circ - im)$, which is the vector addition. Again, it is to be interpreted as $\forall m \in IM,\ P(\sigma, m) = P'(\sigma_i, m) + P''(\sigma_j, m - im)$. The algorithm to perform the well-specified check is presented in Figure 6.4. A model is well-specified if it terminates without encountering **NotWellSpecified**.

6.5.2 Analysis

If $G = (V, E)$ is the configuration graph, then the algorithm performs the well-specified check in $O(V + E)$ time. That is the same amount of time it takes to do a state-space

Generate configuration graph.
P = well-spec-check(root)

algorithm well-spec-check(σ_i) : P
$P = 0$
if $IEN(\sigma_i) = \emptyset$ // stable marking
    $P(\sigma_i, 0) = 1$
    return P
else if σ_i is a case node
    Let $\sigma_i^G = \sigma_i$
    $\forall a \in IEN(\sigma_i) \cap G$
        Let $\sigma_j = a.Complete(\sigma_i)$
        P' =well-spec-check(σ_j)
        $P(\sigma_i, \circ) = P(\sigma_i, \circ) + P'(\sigma_j, \circ - \mathcal{C}(\sigma_i, 0, a, \sigma_j))p_{ij}$
    return P
else if σ_i is a decision node
    Let $G \in AG^r_{C_{\min}}$
    P =well-spec-check(σ_i^G)
    $\forall H \in AG^r_{C_{\min}}, H \neq G$
        if $P \neq$well-spec-check(σ_i^H)
            Not Well Specified
    return P
end algorithm

Figure 6.4: Well-specified algorithm.

generation. In the asymptotic sense, this algorithm for the well-specified check does not add to the state-space generation time, and therefore is asymptotically optimal.

The previous well-specified algorithm performs the check in an amount of time proportional to the number of paths, which can be exponential in E. Thus, the two conditions provide critical insight, allowing us to build a much more efficient algorithm.

However, if one is simulating the model, it is not necessary to explicitly construct the configuration graph in order to generate a sample path. In that sense, the algorithm is not necessarily optimal. Since a simulator does not need to compute the complete distribution of next stable markings, one might be able to use the structure of the model to reduce the amount of work necessary to check whether the model is well-specified. This issue will be addressed in future work.

6.6 Conclusion

The definitions of *well-specified* and *well-defined* are two attempts at managing zero-timed events in stochastic Petri nets and extensions while the model is executing. The well-specified check, adopted in the context of SANs, previously yielded algorithms that have a time complexity that can be exponential in the number of arcs in the configuration graph.

In another context, the definition of *well-defined* was developed. The well-defined check yielded a much more efficient algorithm, having time complexity linear in the number of arcs in the configuration graph. The apparently stricter definition seemed to be the reason a more efficient algorithm could be used.

In this chapter, we modified the definitions to apply to Möbius models. This enabled us to directly compare the two definitions. Next, we introduced two conditions that must be true if a model is well-specified. These two conditions yielded the insight necessary to deduce that the definitions of *well-specified* and *well-defined* are actually equivalent. Consequently, we can use the more efficient algorithm to perform the well-specified check. This algorithm is essentially the well-defined check, and since the asymptotic running time is no more than that of the state-space generator, the algorithm is asymptotically optimal in that context.

CHAPTER 7

OTHER MODEL TYPES

This chapter describes a number of different Möbius model types, including composed models, parameterized models, solved models, and study models. These model types allow users to do more than create atomic models, define measures on them, and solve them. They allow users to construct larger models using reward models as submodels, to explore design spaces more easily by tweaking model parameters, and to use efficient decompositional methods. These model types complete the definition of the Möbius framework and present a complete, coherent, and consistent framework for the development of a modeling tool. Additional model types may also be defined in the future as new needs arise.

Composed models are built using model composition, which is a method for building larger models out of smaller component models called *submodels*. Model composition plays an important part in the Möbius framework, and is more important than one may infer from its simple definition. Model composition is one way Möbius stands out when compared to other modeling methodologies. In Möbius, different model composition techniques may be applied to submodels largly independent of the formalisms used to create the submodels. Furthermore, a composed model may be composed of submodels expressed in different formalisms. This allows for the "mix-and-match" approach, which gives users more flexibility in creating and solving models.

Model parameterization is the process of adding parameters to models and creating a parameterized model. For parameterized models, most model components may be functions of model parameters. Examples of model parameters include the failure rate of a component, or the traffic load of a network. These may be varied to study behavior over a range of conditions. That allows model components to be "tweaked" or "tuned." Parameters are used both in study models and in connected models. They can also be used if the parameter values are not known, and may be computed by some other model, a process known as model *connection*.

Results are simply the computed solutions of reward variables. Specifically, they are computed solutions to the reward variable measures. When results are added to a reward model, they form a solved model. Because results may be used in constructing connected models, as opposed to merely being displayed, results are formally described as part of the Möbius framework.

Model connection is the process of solving a large "overall" model by solving a series of smaller models in isolation and adjusting parameters based on results. This collection of models with sharing of results forms a connected model. The results of one model may be used to compute a parameter value of another model. The dependencies may be cyclic, in which case an iterative solution must be used to find an "overall" solution. The overall "answer" is the computed solution, or results, of the submodels of the connected model.

Study models are a special type of parameterized model; instead of having a single parameter value, the study model has a set of parameter values. The study model is solved for the set of parameter values. Other tools have demonstrated this to be of great practical value, so it is included here.

7.1 Composed Models

7.1.1 Model composition

The purpose of model composition is to make a larger model out of smaller, component submodels. Typically, the models are modified and somehow structurally joined together to make a single, larger, monolithic model.

There are two primary reasons for model composition. First, it is an engineering convenience. Systems are often made of components, and it is convenient to model the system as a collection of components. Thus, each component may be constructed and then assembled (via model composition) into a single model of the larger system. This also promotes the reuse of model components, just as system components are often reused in other systems.

From a technical point of view, an important reason for model connection is that if done in a certain way, it may lead to more efficient solution techniques. For example, if models are replicated (holding certain state variables in common) and then joined with other models, and the reward variables are restricted in a certain way (as is done with the Replicate/Join composition formalism [36]), then the number of states in the underlying state space may be significantly reduced. One particular advantage of this method is that the entire state-space does not need to be constructed; the reduced state-space can be constructed on-the-fly.

One unique capability of Möbius is its ability to compose composed models. Since the

result of model composition is a reward model, the composed reward model may itself be composed. Model solution may take advantage of such hierarchical composition. For example, if one of the submodels of a Kronecker-based approach may be represented as a graph composition model of submodels, the state-space generation of the submodel may take advantage of the symmetries present to produce a reduced-state submodel. Naturally, the Kronecker solution methods must take into account a reduced state space, and in doing so, may reduce the number of symmetries present. However, the composition of composed models has, to our knowledge, never been done before. The Möbius framework not only allow for this, but does so in a general framework.

Most model composition formalisms are based on one of two types of sharing: sharing state or sharing (or synchronizing) on actions. Thus, two models may be composed by sharing (holding in common) a small set of state variables or actions. That way the two models can interact, either by sending messages through state variables, or by synchronizing on certain events. Note that typically, model composition modifies the submodels only slightly. The alterations to the submodel only affect the way the models are connected together.

Another unique feature of Möbius is that the submodels of a composed model may be created by any formalism supported by Möbius, provided that the property requirements of the composed model formalism are met. This is especially useful for large models of heterogeneous systems, in which several formalisms may be used in order to best express certain system types or aspects. This also allows efficient solution methods restricted to certain formalisms to be used in the study of an isolated system component. Later, as the component becomes a part of the larger system and interacts with other components, the possibilities for efficient solution might be reduced (due to the loss of properties), but at least a solution is still possible (e.g., by Monte-Carlo simulation). The state-sharing approach and synchronizing actions are similar in that they share one or a few components. Larger composed models are assembled using modular components, and the composition modifies the constituent models relatively little. This makes the composition process relatively easy for users to use.

Model composition operates on reward models. This is not a restriction, as atomic models can easily be converted to reward models. Let AM be an atomic model. We can create a reward model $RM = (AM, R, RF, \rho_0, RMP)$, where $R = \emptyset$. Thus, the requirement that model composition operate on reward models, and not atomic models, is not a restriction.

Model composition is a mapping:

$$CM : RM_1 \times RM_2 \times \cdots \times RM_k \rightarrow RM\,.$$

Thus, if $RM_1, RM_2, \ldots, RM_k$ is a list of reward models, then model composition is the function $cm(RM_1, RM_2, \ldots, RM_k)$, which maps to a new reward model. Thus, a *composed model* is a reward model that has been constructed by a model composition function.

7.1.2 Example composed model

For example, let RM_1 and RM_2 be two reward models. The state variables in RM_1 and RM_2 are S_1 and S_2, respectively. Let $s_1 \in S_1$ and $s_2 \in S_2$ be state variables of the same type, i.e., $type(s_1) = type(s_2)$.

A model composition function may compose the two reward models by "joining" s_1 and s_2, that is, by making s_1 and s_2 a single, common state variable. Two restrictions on this composition technique are that $|IS_1| = |IS_2| = 1$, and that the initial values for s_1 and s_2 must be the same. (A more relaxed requirement might include a definition of a new IS distribution, but we keep the requirements simple for illustration purposes.)

Let s_c be a new state variable that replaces s_1 and s_2. Thus, the set of state variables in the composed model are given by $(S_1 \setminus \{s_1\}) \cup (S_2 \setminus \{s_2\}) \cup \{s_c\}$. All functions of $\sigma_1(s_1)$ and $\sigma_2(s_2)$ are replaced by a $\sigma_c(s_c)$, where σ_c is the state of the state variables in the composed model. Computer programs may implement this easily by simply changing a pointer or reference. The new model behaves as if s_1 and s_2 became the same state variable, s_c. The actions of the composed model are the union of actions of the submodels, where the action partition group of the composed model is the union of the partition groups of the submodels. Similarly, the reward variables are combined. Properties must be considered individually.

In this example, there may not be any significant benefit from a solution standpoint. However, users may benefit from the use of modularity. Users may also benefit by being able to represent AM_1 in one formalism, and AM_2 in another. This feature is present only in a few frameworks.

It is important that properties be dealt with properly by model composition. Properties state something about a model, but model composition usually changes the model somehow in the process of composing. This change may affect the properties. Ideally, a composition formalism would know the dictionary of properties used by the constituent models, and be able to determine which properties of the submodels are preserved through the composition and apply to the composed model, and which do not. For example, if all the actions in RM_1 and RM_2 have exponential delays and the property <exponential>, then the join composition described above preserves the property. However, if only one submodel has the property, the composed model may not have it.

A composed model may not preserve a property it does not understand. This ensures

correct and conservative behavior of solvers, but may result in lost efficiencies in solution.

7.1.3 Example formalisms

A number of model composition formalisms have been developed. For example, the Replicate/Join formalism of [36] is a composition formalism. In it, models are composed by sharing state, as described in the previous section. The operation of joining several identical models is called "Replicate," and, on-the-fly, the state-space generator may produce a reduced state space if the model is Markov.

This approach has been extended and generalized in the work of [42] on the graph composition formalism. This formalism makes it possible to join models in an arbitrary by by having them share state, and all the model symmetries can be automatically detected. An efficient, polynomial-time algorithm exists to detect the symmetries on-the-fly [42]. It can detect symmetries that exist in a ring or a mesh, for example.

Another approach is based on synchronizing of actions, and has been widely studied in a number of contexts, including stochastic process algebras such as PEPA [44], stochastic automata networks [48], and superposed GSPNs [49], [50]. Efficient solution methods based on these approaches are generally referred to as *Kronecker-based* approaches (e.g., [78]). In these approaches, models are joined together by "synchronizing" on actions. Like state variable sharing, synchronization on actions makes one new action out of two old ones. The new action is enabled if both enabling conditions of the old ones are, and when the action completes, the model changes as if both old actions had completed. The delay characteristics and execution policy depend on the particulars of the formalism.

Kronecker-based methods are efficient if they can compactly store the state space of a model, because the state-transition-rate matrix can be formed one matrix entry at a time on-the-fly as a function of Kronecker operators on the state-transition-rate matrix of the submodels. Numerical solvers therefore do not need to hold the state-transition-rate matrix in memory, since each element of the matrix can be constructed with relatively little computational effort. That can result in a significant decrease in memory requirements, allowing for the solution of larger models with the same amount of computer memory.

7.2 Parameterized Models

Models may be parameterized. This means simply that many of the components of the model may be functions of model parameters. Parameters are real-valued variables that are expected to be varied. Typically, parameters are used for one of two purposes. First, the

parameter may be ranged over a set of values to observe the effect on results. That is done in study models, which are described in Section 7.5. Parameters are also useful when some parameter value is unknown but may be determined by the results of some other model. That is the case in the connected model, which is the topic of Section 7.4.

Parameters are simple model components. They consist of a set of parameters F (one can think of these as "names"), and a set of parameter values $IF : F \to \mathbb{R}$. Parameters are then given by the pair $Parms = (F, IF)$. Parameters are added to various model definitions so that the model components may be functions of parameters.

Recall from Section 3.4 that an atomic model is given by

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

A *parameterized atomic model* is given by

$$PAM = (SV', Act', MP', Parms)\,,$$

where SV' and Act' are state variable and action definitions that have been modified to take parameters into account.

Recall the definition of state variables is $SV = (S, type, IS, P_S, SVP)$, where S is the state variables, *type* is the state variable types, IS is a set of initial states, P_S is a probability distribution over initial states, and SVP is the state variable properties. The parameterized state variable definition is given as $SV' = (S, type, IS', P'_S, SVP')$. Where $IS : 2^{val}$, the parameterized set of initial states is given by $IS' : IF \to 2^{val}$, i.e., the initial states may be functions of the parameters. Similarly, $P_S(v) = \Pr[\text{Initial value of state variables is } v]$, and $P'_S(v, IF) = \Pr[\text{Initial state variable is } v|IF]$; that is, the initial state distribution may be a function of the parameters. Finally, $SVP : S \to 2^{\Pi}$ and $SVP' : S \times IF \to 2^{\Pi}$, i.e., properties may be a function of the parameters. Parameters may alter important model properties; hence, properties may depend on parameters.

Actions are parameterized similarly. Recall $Act = (A, AF, AG, AP)$, where A is the set of actions, AF is the action functions, AG is the action partition groups, and AP is the action properties. Parameterized actions are given as follows. $Act' = (A, AF', AG', AP')$, where A is as before, but each element in AF' may now be a function of the parameter values. For example, $Enabled : \Sigma \to bool$. Now, $Enabled'$, a member of AF', is now given as $Enabled : \Sigma \times IF \to bool$. Action partition groups, $AG : 2^{2^A}$, are parameterized as well: $AG' : IF \to 2^{2^A}$. Finally, action properties, $AP : A \to 2^{\Pi}$, are parameterized as follows: $AP' : A \times IF \to 2^{\Pi}$. Model properties are parameterized similarly.

Reward models may also be parameterized. Recall that a reward model is $RM = (AM, R, RF, \rho_0, RMP)$. If only the reward variables are parameterized (for example, if a reward formalism adds reward variables to a nonparameterized atomic model), then the parameterized reward model is as follows. $PRM = (PAM, R, RF', \rho_0', RMP, Parms)$. PAM is the atomic model that has been trivially parameterized, R is a set of reward variables as before, and RF' is the parameterized reward functions that are parameterized in the same way that action functions are parameterized as discussed above. The term ρ_0 is the parameterized initial state of the reward variable automata, and is a function of the form $\rho_0'(r, fv) \in RF'(r).rt$, where $fv \in IF$. $Parms$ are parameters as defined above.

If the reward variables are defined on parameterized atomic models, then the parameters of the atomic model are moved to the reward variable parameters, and any new reward variable parameters are added. Thus, a parameterized reward model is given as $PRM = (AM'', R, RF', \rho_0', RMP, Parms)$, where AM'' is a parameterized atomic model without any parameters, i.e., $AM'' = (SV', Act', MP')$, made up of parameterized state variables, parameterized actions, and parameterized model properties. The various parameterized atomic model elements are parameters of the $Parms$ element of the parameterized reward model. All else is as before.

Composed models can also be parameterized. Recall $CM : RM^k \rightarrow RM$. Model composition may operate on parameterized reward models, i.e., $CM : PRM^K \rightarrow PRM$. Furthermore, a parameterized composed model is the result of parameterized model composition, i.e., $PCM : PRM^k \times IF \rightarrow PRM$. New parameters may be added in the resulting composed model.

Naturally, given a set of parameter values, a parameterized model is the same as a model as previously defined. Therefore, the question of whether a model is parameterized does not affect most discussions regarding the model. If it matters whether a model is parameterized, we specifically indicate this. It does matter, for example, in model connection (see Section 7.4).

7.3 Solved Models

7.3.1 Introduction

Results are simply the computed solutions of reward variables. They may be computed in any number of ways, including analytically, numerically, or statistically (via simulation), depending on the solution technique. In addition to the computed solution, results include any extra information that the solver may produce. For example, a simulator may statis-

tically estimate the mean, but may also give the estimated confidence interval for a given confidence level. A numerical solver may give an estimation or a bound on the error. This extra information takes the form of "annotations," which are similar to properties in that they are expressed as proprietary symbols.

Solvers operate on reward models and produce solved models. A solved model is simply a reward model coupled with results. We begin by formally describing results.

7.3.2 Elements of results

Let $\mathcal{A}$ be the set of all annotations. We write an annotation as [annotation]. Each result may have an associated set of annotations. Let $An : 2^{\mathcal{A}}$ be a set of annotations.

Recall from Section 4.3.5 that each reward variable has a set of reward variable measures that specifies what the solver should solve for. We restate them here for convenience.

$$\begin{aligned} \mathit{Dist} &: \mathit{bool} \\ \mathit{Moments} &: \mathbb{Z}^+ \\ \mathit{Intervals} &: 2^{\mathcal{B}_{\mathrm{f}}(\mathbb{R})} \end{aligned}$$

Results are designed to provide solutions to each of these reward variable measures. If a reward variable measure is not requested, it is not usually solved for (but may be). If a solver is not capable of solving for some reward variable measure, it provides a null element ν. The elements of results are as follows:

$$\begin{aligned} \text{Dist} &: ((\mathbb{R} \to [0,1]) \cup \{\nu\}) \times An \\ \text{Moments} &: \mathbb{Z}^+ \to (\mathbb{R} \cup \{\nu\}) \times An \\ \text{Intervals} &: \mathcal{B}_{\mathrm{f}}(\mathbb{R}) \to (\mathbb{R} \cup \{\nu\}) \times An \end{aligned}$$

Either the distribution is solved for, or it is not. If it is solved, then the distribution is given as a mapping $\mathbb{R} \to [0,1]$ and An gives any additional information in the form of annotations. If the distribution is not solved for, then the distribution is the element ν, and the annotations may give some additional information, perhaps regarding why the distribution was not solved. Similarly, for each moment and set of intervals, either a solution is presented (in the form of a real number) or it is not (ν), and each moment may also have annotations.

7.3.3 Solved model

Results for a model are the combination of all of the results for all of the reward variables. Results may also contain annotations. Annotations may be used, for example, to express the number of iterations a numerical solver uses, or what stopping criterion was reached, or the number of experiments executed.

Formally, we can write the results of a model as follows.

$$Res : R \rightarrow \text{Dist} \times \text{Moments} \times \text{Intervals}\,,$$

where R is the reward variables of a reward model. A reward model is considered *solvable* if $R \neq \emptyset$.

A *solved model* is then a reward model coupled with results, i.e.,

$$SM = (RM, Res, SMA)\,,$$

where $SMA : 2^{\mathcal{A}}$ is the annotations of the solved model. Examples of solved model annotations include the number of replications of a simulation, or the number of iterations of a solver. The solved model annotations apply to all the reward variables, as opposed to a particular reward variable.

A *solver* is a module that takes a reward model and produces a solved model, i.e.,

$$solver(RM) \mapsto (RM, Res, SMA)\,,$$

where Res is the results of the reward model RM.

7.4 Connected Models

7.4.1 Introduction

A connected model is essentially a collection of models that communicate by exchanging results. In contrast, composed models typically interact by sharing state or synchronizing on actions. One composed model may send a message to another composed model by altering the value of a state variable or enabling an action. Connected models also send messages, but at a higher level of abstraction. A connected submodel is solved in isolation. The results of the submodel are then used to compute parameter values of other submodels in the connected model. If RM_i is solved and shares results with RM_j, then in essence a new

RM_j model is constructed that is identical to the old RM_j except that certain parameter values may be altered. RM_j must then be solved with the new parameter values. Instead of sharing state or actions like composed models do, connected submodels share results.

In order to solve a connected model, it is necessary to solve each of the submodels at least once, and perhaps to solve submodels iteratively with different parameter values until some stopping criterion is met. Each model is parameterized, and the parameter values of some or all of the submodels are defined in terms of results of the other submodels.

Connection is often used in decompositional techniques, e.g., [79], [80]. Typically, an initial guess at the parameter values is given; then models are solved iteratively until some overall solution is computed. It may be that no overall solution exists. Möbius does not address the issue of existence or uniqueness of a solution, e.g., see [81]. It simply provides the facilities to perform model composition, and leaves those issues to solvers or users.

7.4.2 Connection elements

A connected model is made up of four basic components. The first consists of a collection of solvable, parameterized models. These are the models that share results. The second is a relation called the "shares-with" relation, which precisely describes how certain models share results with each other. The third is a function called the "sharing" function, which describes how results are used to compute model parameters. The fourth is a stopping criterion, which determines when the overall solution is sufficient.

More formally, let $PRM_1, PRM_2, \ldots, PRM_K$ be a collection of parameterized reward models that make up the connected model. We call RM_i a *submodel* of the connected model. An irreflexive relation called the *shares-with* relation is used to describe the sharing relation between submodels. We write the shares-with relation $\xrightarrow{*}$, and the model pair (RM_i, RM_j) belongs to the relation $\xrightarrow{*}$ if the results of RM_i are used to compute the parameter values of RM_j. We write $RM_i \xrightarrow{*} RM_j$.

Next, we define the *sharing* function. Let $sharing_i : Res^{(i)} \rightarrow IF_i$, where $Res^{(i)} = \{Res_j : RM_j \xrightarrow{*} RM_i\}$. Then *sharing* is the collection of $sharing_i$, for $i = 1, \ldots, K$, i.e., $sharing = \{sharing_i\}$.

The rationale for having a shares-with relation is purely for solution efficiency. Computing the solution to a model may be computationally expensive, and the shares-with relation may reduce the number of model solutions needed, especially if the shares-with relation forms a graph that is not strongly connected. In graphs that are not strongly connected, the reward models that are not strongly connected may only need to be solved once. Furthermore, many model connection schemes use the shares-with dependency to determine the order in

which models should be computed. The actual order in which models are solved, and the parameter values that are used are determined by the model connection formalism.

A connected model is considered "solved," that is, the results are sufficient, if some stopping criterion is reached. The stopping function $stop : \circ \rightarrow bool$ is *true* when the stopping criterion is reached. The domain of the function is left undefined, as it can be a function of virtually anything. Typically, it uses the current and possibly the past results to determine when the stopping condition has been reached. For example, the stopping criterion may be true if $||Res^{(i+1)} - Res^{(i)}|| < \epsilon$, where $Res^{(i)}$ is the ith result in some iterative solution method. For example, a connection formalism may seek a global or local optimum for some result value. All the result values may be used to determine whether an optimum is sufficiently "good" and stop.

Finally, we can define a connected model as follows.

$$ConM = (PRM^1, \ldots, PRM^K, \stackrel{*}{\rightarrow}, sharing, stop)$$

7.5 Study Models

The Möbius tool includes the concept of a "study." While not of much theoretical interest, it is very useful in practice. A *study model* is a parameterized model with a set of parameter values. The intent is that the study model will be solved for each of the parameter values.

Study models are useful in practice for a number of reasons. First, in the design process, some parameter values may not be known. Studies are particularly useful in the design process in exploring the design space so that trade-offs can be studied, which may then lead to better design decisions. For example, one can design a component to be more reliable, or use redundant components. The best choice may be a complex function of the dependability, performance, and cost. A search of the design space may help us find the best trade-offs. This ability has been found very useful in other tools, including *UltraSAN*.

A study model differs from a parameterized reward model only in that it includes a set of parameter values instead of a single parameter value. Let F be a set of parameters as before. A *parameter study* is the set of mappings $FS : 2^{F \rightarrow \mathbb{R}}$. Let study parameters be defined as $SParms = (F, FS)$. Then we can define a (parameterized) study model as

$$PSM = (AM', R, RF', \rho_0', RMP, SParms)\,.$$

Study-connected models may also be constructed. Some parameters may be used to search a design space, and other parameters may be used for model connection. A study-

connected model is a connected model constructed from a collection of study models. The parameter values in the study model are used as the initial parameter values in a set of connected models.

7.6 Conclusion

We have shown a number of extensions to the atomic and reward models. We defined the composed model, which is simply a "joining" of a number of reward models to form a single, larger reward model. Model composition may increase modeling convenience by providing modularity, or it may aid in efficient solution, such as by on-the-fly state lumping or Kronecker-based solution methods.

We showed how atomic and reward models may be parameterized. Parameterization may also be used for modeling convenience, as in study models, or to aid in efficient model solution using connected models. We defined results as the computed solution to a reward variable measure plus any additional information in the form of annotations. Using results, we are able to define connected models.

We described connected models, which are sets of reward models that "communicate" by sharing results. The result of one model may be used to compute the (input) parameter values of other models. Typically, the models are solved iteratively to form an overall solution. We do not address the many issues relating to connected models, especially existence and uniqueness of solutions; we leave that to connected model formalisms, and simply provide the mechanism for doing model connection.

Finally, we describe a minor but useful extension to parameterized models, namely the study model. A study model allows users to solve models for a set of parameter values instead of a single parameter value. The study model is of practical interest, but does little to the theoretical framework. We include it because of its practical usefulness, and as an example of the extensibility of the Möbius framework. We also introduced the study-connected model.

The model types described in this chapter complete the formal specification of the Möbius framework.

CHAPTER 8

DEF EXAMPLE FORMALISM

8.1 Introduction

Many of the concepts we developed in the Möbius framework are difficult to illustrate, because no formalism exists that can take advantage of most or all of the features available. It is not necessarily the case that a feature-laden formalism is a good and useful formalism. Many formalisms are useful because of their limitations, especially insofar as their restrictions increase solution efficiency. However, we believe that a formalism can take advantage of most of the features of Möbius and be very useful.

In this chapter, we develop a new formalism called DEF (Dan's Example Formalism). We have two primary intentions for this formalism. First, we designed it for the purpose of exposing most of the features available in the framework. The formalism has a moderately direct mapping between the formalism components and the framework components. The general philosophy behind DEF is that we wanted it to look as if the Möbius framework definitions were translated to a high-level object-oriented programming language. State variables are represented by variables, while actions and reward variables are special objects known to the language; objects can be derived and instantiated to form particular actions and reward variables. While DEF is a complex, verbose language, we have spent some effort making it easy to express simple things with it.

Our other objective for DEF was for it to actually be a useful modeling language with which complex models could easily be constructed. We targeted two solution methods for DEF: Markov process-based analytical methods, and Monte-Carlo simulation. DEF is perhaps best described as a hybrid between stochastic Petri nets and simulation languages. It has most of the expressiveness that is found only in simulation languages, but also includes many space-saving constructs to make state-space generation efficient. It was designed to fill the niche of the need to express models that are too difficult or tedious to construct using

stochastic Petri nets, but can still be solved by Markov process-based solution methods. It also offers a model migration path. Simple, analytical models can evolve into larger, detailed models for which simulation is the only available solution method. This can be done without any need to change formalisms.

We paid special attention to the problems of model correctness. Just as it is difficult to write a correct program, and it is certainly very hard to write a large, correct program, it is also hard to construct a correct model; it is increasingly hard as the model grows in size and complexity. While there is no substitute for a disciplined construction process, we included facilities in the language that we believe will aid in constructing correct models. Strong type-checking and runtime error detection detect many common errors, and assertions and traces facilitate validation and debugging. When the validation process is complete and performance is important, the runtime checks and assertions can be removed to achieve, we believe, C++-like performance. Unfortunately, to date no prototype of this language has been built, so we can only make educated guesses about what the performance of an implementation would be.

DEF is based on Pascal with object-oriented extensions. Only important object-oriented extensions are included. (Future extensions of the language may incorporate more sophisticated object-oriented features.) Users who are familiar with Pascal or any similar high-level language will find DEF easy to learn. DEF does make some significant changes to Pascal. For example, we added new basic types to facilitate actions and multisets. Semicolons are now used as terminators (as in C), not as separators (as in Pascal). Also, we added procedure and method modifiers that are necessary for defining actions in DEF.

8.2 Data Types

In order to use efficient Markov-process-based solution methods, we designed DEF such that each state variable need only take the minimum amount of memory and so that the amount of memory used by all the state variables is constant. Thus, we avoid constructs that can lead to state variables whose sizes are unknown, or to an a variable number of state variables. Unfortunately, this means that can not use dynamic memory allocation in our language.

DEF borrows many aspects of high-level programming languages, notably Pascal. Pascal has many of the features that we desire in a formalism. For example, Pascal is a strongly typed language. This allows for strong type checking. Since DEF is a strongly typed language, we encourage the use of named types whenever possible. By convention, we declare a

type name for any type that we use that does not already have a name. Pascal also has many of the space-saving features that we desire, such as subranges (e.g., `5..10`) and enumerated types.

8.2.1 DEF data constructs

In this section, we describe the various DEF variable types, and compare them to Möbius state variable types. (Refer to Figure 3.1 for the construction rules for state variable types.) Most of the constructs follow standard Pascal language constructs.

Subrange

We begin by describing the subrange. We adopt the standard Pascal subrange notation. For example, if we want to declare a state variable `x` that can take on values in the set $\{1, \ldots, 20\}$, we can do so in following way:

```
var  x : 1..20;
```

There are a number of advantages to declaring a variable this way. First, it is clear from the definition that `x` may only take on values in the range from 1 to 20. This is usually better than simply declaring `x` to be an integer. Using subranges also allows for runtime range checking. If, for some reason, `x` is assigned a value outside the allowable range, the system can immediately detect the problem and declare an error. Finally, because we have explicitly put a range on `x`, a state-space generator need only allocate 5 bits to represent the value of `x`. This has advantages over Petri net automata, for example, in which the number of tokens might not be known in advance, so that some arbitrarily large maximum must be assumed.

Enumerated types

Another frequently useful construct is the use of enumerated types (sometimes called *scalars*). They also follow the standard Pascal syntax. Here is a small example:

```
TYPE
  ws_state = ( idle, busy, rebooting, failed );
VAR
  computer : ws_state;
```

This example declares a variable called `computer` that can take on one of the four named values. The default initial value of `computer` is idle. Use of enumerated types is encouraged

both for user readability and for efficiency reasons, as with to subranges.

One disadvantage of working with stochastic Petri nets is that tokens have no inherent meaning. While this offers a great deal of flexibility, it can make models difficult to read and understand. For example, a place with n tokens may represent n "things," or it may represent a "thing" of type n. Enumerated types make the meaning clear because the number n is given a meaningful name. (In fact, the value of n is implicit.)

Predefined types

DEF has only a few automatically defined types. We encourage people to declare their own types and not use basic types for state variables. However, DEF does provide many of the same basic types that Pascal does, namely integers, reals, Booleans, and chars. (We omit complex types because of their marginal usefulness in a modeling context.)

The integer type follows the standard Pascal definition. A constant identifier called `maxint` is defined, and the value of an integer is guaranteed to be between `-maxchar` and `maxchar`. Typically, on 32-bit microprocessors, the value of `maxchar` is 2^31-1, or 2147483647. Thus, we can define the integer as follows:

```
TYPE integer = -maxchar .. maxchar;
```

The `real` type is the IEEE floating point representation. Users are discouraged from using this type, primarily because certain mathematical identities may not hold (because of round-off errors). For example, $\sum_{i=1}^{7} \frac{1}{7} = 1$, but the sum is not 1 when performed with IEEE arithmetic. Again, consistent with Pascal, constant identifiers `minreal`, `maxreal`, and `epsreal` are provided. We omit the description here for brevity. In addition, DEF provides `infinity`.

The `char` type is again as defined by Pascal. It is not a small integer, like in C, but rather a special type that takes on character values between 0 and `maxchar`. The character values must be a letter enclosed in single quotes, e.g., `'a'`.

Boolean values can be especially useful. They are predefined by the language the following way:

```
TYPE boolean = ( false, true );
```

Thus, the ordinal value of `false` is 0 and `true` is 1.

State variables of basic types are declared in the same as one would declare a global variable in Pascal. Here are some examples:

```
VAR
```

```
TYPE
   ws_state   = (idle, busy, rebooting, failed);
   Rcomputer = record
      state       : ws_state;
      free_memory : integer value 64;
   end; (* Rcomputer *)

VAR
   computer : Rcomputer;
```

Figure 8.1: Example use of `record`.

```
  x    : integer;
  y    : boolean;
```

The initial value of these variables is 0. That is the default. We can override the default initial value by using the `value` modifier. For example,

```
VAR
  x    : integer value 5;
  y    : boolean value true;
  c, d : char value 'w';
  x    : real value 3.141593;
```

Records

Records are also supported via the `record` construct. Again, we follow the standard Pascal notation. By convention we always give our records type names. Figure 8.1 illustrates the use of a `record`.

Records with variant parts are a useful extension to records. Observe the Pascal notation illustrated in Figure 8.2. In this example, we presume that many of the types have been previously declared, and focus only on type `packet`. The record has a fixed part with a variable called `tag` of type `Tpacket` (assumed to have been declared earlier). Every variable of type `packet` has access to this element. Depending on the value of `tag`, the rest of the record may have one of two different structures. If `tag = data`, then the record has elements `source`, `dest`, and `length`. Otherwise, if `tag = control`, the record has elements `dest` and `info`. Note that as in Pascal, variant parts may be nested.

Frequently, records with variant parts can be eliminated through the use of object-oriented methods; that usually results in more compact, readable, and maintainable models.

```
. . .
Tpacket = record
   case tag : Tpacket of
      data : (
         source, dest : node_id;
         length : Tdatalength
      );
      control : (
         dest : node_id;
         info : Tcontrolinfo;
      );
end;
```

Figure 8.2: Example record with a variant part.

However, there are some legitimate reasons for using records with variant parts, so we provide them.

Arrays

Arrays are also a DEF type. Arrays are declared the same way that they are in Pascal, as the following illustrates:

```
 TYPE
   ws_state = (idle, busy, rebooting, failed);
VAR
   computers : array[1..9] of ws_state;
procedure dosomething;
   begin computers[5] := rebooting; end;
```

It is possible to assign an array value to an array using the extended Pascal syntax of an array value. That is useful for assigning initial values to arrays, as the following illustrates:

```
TYPE
   ws_state = (idle, busy, rebooting, failed);
VAR
   computers : array[1..9] of ws_state value
      [1,2 : idle; 3..5, 7 : busy; otherwise failed; ];
procedure failall;
   begin
   computers = [1..9 : failed];
```

Table 8.1: DEF set symbols.

Meaning	Mathematical symbol	DEF symbol
Set	$\{\ldots\}$	`[...]`
Empty set	$\emptyset$ or $\{\}$	`[]`
Union	$A \cup B$	`A + B`
Intersection	$A \cap B$	`A * B`
Inclusion	$a \in B$	`a IN B`
Set equality	$A = B$	`A = B`
Set inequality	$A \neq B$	`A <> B`
Set difference	$A \setminus B$	`A - B`
Contains	$A \subseteq B$	`A >= B`
Is contained by	$A \supseteq B$	`A <= B`

```
end;
```

Sets

DEF implements the `set` construct as defined in Pascal. Table 8.1 gives a listing of frequently used math operators on sets, such as union, set equality, and inclusion, along with their corresponding notations in Pascal. Some of these are standard Pascal syntax, some are extended, and some correspond to other variants of Pascal. Since sets may be used frequently for state-space reduction, we provide a rich set of operators in DEF.

Like most Pascal implementations of sets, DEF implements sets as bit vectors. Thus, the type of a set should be a simple type with relatively few unique values. DEF arbitrarily restricts the number of different set elements to be 256. Thus, a set of `T` requires that `T` have a cardinality of 256 or less. (This requirement is common to many Pascal compilers.)

Since sets may be particularly useful for reducing both the size of a state representation and the number of states in the state space, we considered augmenting sets so that they may contain a maximum number of elements of any type. That would allow us to relax the cardinality restrictions on a set of `T`. However, we decided against this. It would require a significant change to the language and syntax, and it is not clear that the extension would be useful. Also, this functionality could easily be realized using the object-oriented extensions that we discuss later.

Multisets

Another powerful construct we have in DEF is the multiset. The multiset declaration includes the minimum and maximum allowable number of any element in the multiset. If

Table 8.2: DEF use of multisets.

Meaning	Mathematical symbol	DEF symbol
Summation	$C = A + B$	`C = A + B`
Scalar multiplication	$C = n \times A$	`C = n * A`
Relations	$A = B$	`A = B`
Size of	$\|A\|$	`abs(A)`
Cardinality	$n = A(s)$	`n = s IN A`
Subtraction	$C = A - B$	`C = A - B`

only an integer is given, that number is assumed to be the maximum, and the minimum is assumed to be zero. An example declaration of a multiset follows:

```
TYPE
  ws_state = (idle, busy, rebooting, failed);
VAR
  A, B, C : multiset [0..16] of ws_state;
```

Table 8.2 illustrates how multisets may be used.

All the relational operators ($=$, $\neq$, $<$, $\leq$, $>$, and $\geq$) look like their scalar counterparts and are true if the relation holds for each element in the multiset. Note that multiset subtraction `C = A - B` may occur only if `A >= B`; otherwise, the operation results in an error. The `abs()` function gives the sum of the number of elements in the multiset. Note that an element may have a negative cardinality, so the size of a multiset may be negative.

If a set and a multiset are defined over the same type, then the multiset is assignment-equivalent to the set, that is, it is possible to assign the set to the multiset. Furthermore, it is possible to promote the set to a multiset for the purpose of doing scalar multiplication. This is useful for assigning a literal value to a multiset. Consider the following example:

```
TYPE
  tenrange = 1..10;
VAR
   s : set of tenrange;
   m : multiset[5] of tenrange;
begin
   s = [1, 7, 8];
   m = 3 * s;
   m = m + 2 * [5, 6] + 2 * [8];
   ...
```

Here, we define a set over `tenrange` and a multiset over `tenrange` that takes on cardinalities of `[0..5]`. The first line assigns a value to the set `s`. The second line promotes `s` to a multiset and does a scalar multiplication by 3. After the third line, element 1 of `m` has cardinality 3, elements 5 and 6 have cardinality 2, and element 8 has cardinality 5. All other elements have cardinality 0.

The use of sets and multisets is a valuable tool for solvers that enumerate the state space, and we are able to use the information about how the state variable is constructed in order to take advantage of possible efficiencies, both in reducing the number of states and in reducing the memory requirements necessary to represent a state. For multiset `m`, we have at most 10 distinct elements in the multiset, each with a range of 6 different cardinalities. Thus, the state of `m` can be expressed in $10 \times 4 = 40$ bits.

Pointers

Next, we introduce the pointer construct. Note that our pointers differ somewhat from those defined in Pascal. A Pascal pointer can be assigned a value only through the use of the `new()` function. In DEF, we do not use `new()`. However, pointers, especially pointers to objects, are useful. Consider a packet moving through a network. The packet may be a large structure, and each node in the network may have a number of buffers. Instead of copying the packet to each node, we can use a pointer and change the pointer value instead. While this does not affect the number of states, it has the potential to substantially decrease the size of the representation of a state.

The following illustrates the use of pointers:

```
 TYPE
  ws_state = (idle, busy, rebooting, failed);
  ws_state_ptr = ^ws_state;
VAR
  computer : ws_state;
  next : ws_state_ptr value @computer;
procedure rebootNext;
   begin
   next^ := rebooting;
   end;
```

Note the use of the `@` operator on `computer`. It is the "address-of" operator. Thus, the initial value of `next` is assigned the address of `computer`. The procedure `rebootNext` assigns the value `rebooting` to the `ws_state` pointed to by `next`.

Examples in DEF

We illustrate the use of DEF on some simple queues. We begin with a simple, closed queueing network with three customer classes.

First, if the queueing discipline of a queue is FIFO (first come first serve), then the order in which customers arrive is important and must be preserved. The queue is made up of an array of slots, with each slot being either empty or occupied by a customer of class 1, 2, or 3. Thus, we can write the following:

```
TYPE
  Tslot = (empty, class1, class2, class3);
VAR
  queue1 = array[1..16] of Tslot;
```

One can observe that again a slot may have only four different states, so each state can be represented by two bits. The number of bits necessary to represent `queue1` is $2 \times 16 = 32$ bits. Again, this efficiency can be gained by using the structure naturally available to the DEF formalism editor.

The number of possible configurations that this state variable can assume can be computed. A rough overestimate of the number of states of this state variable can be computed as $4^{16} = 4\ 294\ 967\ 296$. Not all of those states are possible, however. An accurate count of the number of states yields $1 + 3 + 3^2 + 3^3 + \cdots + 3^{16} = 64\ 570\ 081$.

Now consider a priority queue. Customers of class 1 are always served before all other customers, and class 2 customers are served before class 3 customers. It is not necessary to distinguish among customers of the same class. Only the number of customers of a certain class need to be known. We can use a multiset to gain efficiency.

```
TYPE
  Tclass = (class1, class2, class3);
VAR
  queue2 : multiset[0..16] of Tclass;
```

Again, we declare a state variable with the name `queue2` that has as many as 16 elements. In this example, we say that the queue is full if the multiplicity of the multiset (`abs(queue2)`) is 16, and we do not let any more customers in. This approach is useful when enumerating the state space. Since there are 17 possible multiplicity values, each may be represented in 5 bits, and the state variable can be represented using $5 \times 3 = 15$ bits. The number of possible configurations for this state variable is $\binom{19}{2} = 342$. One can see that the amount of memory

needed to represent the state variable can be reduced, and that the number of states can be greatly reduced, if we can use the multiset construct instead of the array construct.

8.2.2 Initial value distribution

DEF uses a simple construct to specify the distribution of the initial values. By default, integers take on value zero, subranges take on the smallest allowable value, Booleans are false, scalars take on the value of the first element, sets are empty, multisets have zero or the smallest positive cardinality, and pointer values take on `NIL`. These default values may of course be overridden using the `value` keyword, e.g., `VAR x : integer value 5;`.

Extended Pascal has a construct that allows the initial state of program variables to be set using the following construct.

`to begin do` *statement* `;`

We extend this in two ways. First, we expand the construct to be

`to begin do` *block* `;`

This makes it possible to declare and use local constants, types, and variables in a `to begin` section. This makes the process of setting initial model variables more like a procedure definition and allows for greater flexibility. Setting the initial state is important in DEF, and DEF allows complex initial values to be set only using this construct, unlike Pascal programs, which can have an initialization section of code. Thus, DEF justifies a more powerful `to begin` construct, and since this construct is treated much like a procedure, there is little extra work involved in supporting it.

In order to facilitate a probability distribution of initial states, we introduce a new construct called `with probability`. The `with probability` construct is similar in syntax to the `case` construct in that it may have many parts. The syntax is as follows:

`with probability`
    real-expression `:` *statement*`;`
    ...
    real-expression `:` *statement*`;`
    `otherwise` *statement*;
`end;`

The `otherwise` portion is optional. A *real-expression* is any real-valued expression that takes on values in the range $[0, 1]$. Naturally, the value of each *real-expression* must be between

0 and 1, and the sum of the real-valued expressions must not exceed 1, or an error occurs. If the sum is less than 1 and and an `otherwise` is provided, then the remaining probability mass is assigned to the statement following the `otherwise`. If no `otherwise` is provided, then the remaining mass is assigned to the default value.

Nested `with probability` statements may be used. The probability that the statement will be executed is the product of all the real-valued expressions for each of all of the nested cases.

8.3 Procedures and Functions

Just like Pascal, DEF supports functions and procedures. Both procedures and functions are declared in the same way they are in Pascal. We assume that the reader is familiar with Pascal syntax; the syntax for DEF functions is similar. The only difference is that DEF functions use modifiers, which are discussed in Section 8.4.2.

8.3.1 Provided functions

Most of the functions provided by Pascal are provided by DEF. File-handling procedures and functions are included, but with some restrictions. Procedures that read from a file may only occur during the initialization section, because the state of the system must be entirely contained in the model. File writing may occur at any time. During initialization, loading some state information from disk is allowable. However, if it was possible to read from disk during model execution, some of the model state might be on disk. That would negate the use of most (if not all) solvers. File writing, however, is allowed at any time. This may be useful for any number of reasons. A user may wish to write out information in order to aid in model validation, for example.

String procedures are important for file I/O. Therefore, they are supported. They are described in [82].

The `halt` procedure will be implemented in DEF. The result of executing the `halt` procedure will be an abortion of the model execution and optionally printing some information about the state of the model. This feature will be useful for debugging models. If the `halt` instruction is executed, it may be presumed that an error has occurred.

Most arithmetic functions that are provided in Pascal are also provided in DEF. Most transfer (e.g., `trunc` and `round`) and ordinal functions (e.g., `chr` and `succ`) are also supported.

Unsupported features

A number of features of the Pascal language are there because Pascal is designed to be a programming language. Since DEF is a modeling language, some of the procedures and functions in Pascal are not useful to us.

Dynamic allocation procedures are not included in DEF. The reason is that in DEF, the size of the state of the model is constant. Using dynamic allocation would violate that design feature. Thus, `new` and `dispose` are not supported in DEF.

The use of `pack` and `unpack` is unnecessary in DEF because packed variables are unnecessary. The implementation automatically takes care of packing whenever it is necessary, and this process is invisible to the modeler. The modeler has the convenience of using unpacked structures and, whenever necessary, the benefit of packed structures, which DEF automatically takes care of.

Time procedures are unnecessary in DEF. While it may be useful to write a timestamp to a file, it is also possible that the time may be used to make a decision within the model, so that the time would become part of the state of the model. To avoid that possibility, we eliminate `GetTimeStamp`, `date`, and `time`.

8.3.2 Assertions

One of the facilities of DEF that aids in the production of correct models is the use of assertions. The basic assertion is formed as follows:

`assert (` *boolean-expression* `) ;`

The assertion should assert that `boolean-expression` is true, and that an error occurs if it isn't. If an error occurs, DEF provides a meaningful error message and stops program execution. The `assert` function is roughly equivalent to the following:

```
procedure assert(b : boolean);
   if (b) then halt;
```

Assertions are included as an aid in the verification of models. Assertions may be "turned off" with an appropriate switch once the model has been validated and execution speed is important.

The following illustrates a simple use of assertions:

```
function sqrt(x : real) : real;
   begin
   assert(x >= 0);
```

```
. . . (* Calculate square root of x *)
assert(sqrt**2 == x);
```

Note that round-off errors may cause this assertion to fail. A better assertion would be to check if the square of the root is sufficiently close to `x`.

8.4 Objects

DEF is an object-oriented language that is loosely based on object Pascal and other object-oriented extensions to Pascal. Only the most important and useful the features of a robust object-oriented language are included in DEF. Remember that DEF is a modeling language and not a programming language, and thus DEF also has some extensions that are not a part of most object-oriented languages. In this section, we describe the object-oriented features of DEF.

8.4.1 Classes

An object is an instantiation of a class, which is a special kind of type. Classes are similar to records in that they may contain data fields. Classes are different in that they may not contain a variant part and they may contain methods, which are functions and procedures. One of the key features of any object-oriented language is that the data and the methods that operate on the data may be packaged together in a single unit.

DEF uses the reference model for objects. A class definition defines the type of an object. A variable definition of a class type declares a reference to the object. The object is not actually created until a constructor is called.

A class is declared much like a record except, that it may have function and method declarations. Below is an example:

```
TYPE
   TCircle = class
      x, y, radius : real;
      constructor Create;
      procedure SetCoord(xx, yy : real);
      procedure SetRadius(rr : real);
      function GetArea : real;
      procedure Draw;
   end;
```

An object may be declared to be a variable of type `TCircle`. The following illustrates how a circle may be declared and used:

```
VAR circle1, circle2 : TCircle;

to begin do begin
   circle1.Create; circle2.Create;
end;

procedure dosomething;
   begin
   circle1.SetCoord(10, 20);
   . . .
```

Once a class with methods is declared, each method must be declared. This is done in the standard way.

```
procedure TCircle.SetCoord(xx, yy : real);
   begin
   x := xx;
   y := yy;
   end;
```

Normal scoping rules apply.

Function declarations frequently return values that that are expressed as simple expressions. That is the case in our circle example. The definition of the `GetArea` function is a simple expression. For convenience, we allow an alternative way to declare this function.

```
function GetRadius : real := 3.141593 * radius ** 2;
```

This is a trivial extension, and since we frequently define functions to be simple expressions (especially for actions and measures), the convenience is worth the extension of the language.

Class definitions may occur only at the global level. Thus, classes may not be defined within a procedure, function, method, or constructor.

8.4.2 Modifiers

A modifier is placed immediately after the procedure, function, or method declaration that it modifies. A modifier may also be used to modify a class. An example modifier is the `pure` modifier, which can be used on a function that does have any side effects.

```
function sqrt(r : real) : real; pure;
   . . .
```

It is clear that the function `sqrt` does not modify the state of anything, but merely returns the square root of the parameter. The `pure` modifier enforces this restriction. Note that a pure function may declare some local variables and modify them. This is acceptable, because when the function returns, the life of the local variables ends, and the system state has not changed.

Modifiers are like adjectives that give some additional information about the class or method they modify. Most modifiers can be used in conjunction with other modifiers. Some can only be used on classes; and others only on methods.

Abstract

Classes and methods can both be abstract. An abstract class is a class that can not be instantiated, even if all the methods are defined. An abstract method is one that may not have a definition, and a class that has any abstract methods is also abstract. Only derived classes that override any abstract methods may be instantiated. A class that is derived from an abstract class and that does not override an abstract method is itself an abstract class, even if it is not declared to be so. The following illustrates an abstract method and class:

```
TYPE
   TShape = abstract class
      procedure Draw; abstract;
   end;
```

Override

Any method or constructor can be overridden in a derived class. Abstract methods must be overridden for a class to be instantiated. For example, if we wish to derive a circle from a shape, we will want to override the `Draw` procedure. This is done using the override directive.

```
   TYPE TCircle = class (TShape)
      procedure Draw; override;
   end;
```

Only a member of a parent class may be overridden. An overridden method must have the same spelling and parameters as the method in the parent class.

`Pure`, `Local`, and `Protected`

The `pure` modifier applies to methods, procedures, and functions. A pure method, procedure, or function may not affect the state of a DEF model. It may declare local, temporary variables and modify them, but no other variables may be modified. A pure procedure does not make sense, because it can not do anything.

A `local` method may only modify the state of the associated object. No other object state or global state may be modified. Thus, local methods can only call pure functions, pure methods, or local methods on the associated object. A local procedure or function can only modify global variables, and can only call pure methods and functions or other local functions.

A `protected` method may not modify the state of the associated object. It may modify the state of any other object or global state. Protected methods may call pure methods or pure or local procedures and functions. Protected procedures and functions may call pure or protected procedures and functions, or pure methods.

Variables that are declared `protected` may not be modified after initialization. Thus, they act as constants after the model is initialized. Data members that are declared `protected` may only be modified by the constructor.

8.4.3 Constructors

A reference to an object may be declared in the variable declaration section, but the object is not instantiated until a constructor is called on the reference. The constructor is typically called in the initialization section. A number of different constructors may be defined for each class, but only one may be called to instantiate the class.

The constructor is a special type of method that is like a procedure. It is declared much like a procedure is. See Figure 8.3 for an example. Before the constructor is called, the object reference has a value of `Nil`. Referencing a method or field of a `Nil` reference results in an error.

A constructor must be called for an object before the object can be used. Before the constructor is called, the object reference has a value of `Nil`. Referencing a method or field of a `Nil` reference results in an error.

Note that DEF does not support destructors, because destructors are not strictly necessary.

```
TYPE TCircle = class
   constructor Create;
   constructor Init(xx, yy, r : real);
end;

VAR circle1 : TCircle;

constructor TCircle.Create;
   begin x := 0; y := 0; radius := 0; end;
constructor TCircle:Init(xx, yy, r : real);
   begin x := xx; y := yy; radius := r; end;

to begin do
   circle1.Init(10, 20, 5);
   . . .
end;
```

Figure 8.3: Example use of constructors.

Inheritance and polymorphism

A class in DEF supports the usual features of inheritance and polymorphism. DEF does not currently support many of the advanced features of object Pascal, such as views and properties.

A class may inherit all attributes (i.e., fields, methods, and constructors) of another class:

```
TYPE
   TShape = abstract class
      x, y : real;
      constructor Create;
      procedure Draw; abstract;
   end;

   TCircle = class (TShape)
      radius : r;
      constructor Create; override;
      procedure Draw; override;
   end;
```

DEF supports polymorphism, which is essentially the ability to create a class that inherits from multiple classes. For example, a *spork* is an eating utensil that resembles both a spoon

and a fork. If spoon and fork were both DEF classes, we could define a spork this way:

```
TYPE Tspork = class (Tspoon, Tfork)
   ...  (* New declarations *)
   end;
```

A class can be inherited only once. The order of the base classes is irrelevant. A class can not have itself as an ancestor, or inherit the same class from two or more classes. Inheritance can add new fields and members, but it cannot remove any from the parent classes.

It is not possible for a class to inherit from multiple parent classes if the parent classes have members with the same name. A parent may not be inherited more than once. Thus, it is not possible to define `Tspork` if `Tspoon` and `Tfork` are derived from a common class `Utensil`.

8.4.4 The `Root` class

Every concrete object in DEF is inherited from a special class called `Root`. This inheritance need not (and usually should not) be explicitly stated when a new class is being declared. The root class allows every class to have some minimum functionality. The declaration of `Root` is given below:

```
TYPE
   Root = abstract class
      constructor Create;
      function    Clone : Root;
      function    Equal(r : Root) : boolean;
   end;
```

The `Create` constructor is intended to create an object of the class. The default behavior is to do nothing.

The `Clone` function is supposed to return a copy of itself. Again, the default behavior is to do nothing. If a user wishes to use `Clone`, then he must define `Clone` on all classes. Note that in DEF, `Clone` can only be called in the initialization section.

The function `Equal` is intended to provide some notion of object equality. Its default behavior is to return `true` if the two objects are the same objects. This method is used by the state-space generator to determine whether two objects are equal. Special care should be taken to ensure correctness when defining this method.

8.5 Actions in DEF

8.5.1 Definitions

Actions in DEF are represented as objects. There are two classes from which actions may be derived: `AAction` and `Action`. The `AAction` is the most abstract version of an action, in which every method is defined as abstract. `Action` is derived from `AAction` and provides default behaviors for most action methods. Figure 8.4 shows these two class declarations and definitions. Finally, a zero-timed action called `ZAction` is defined in Figure 8.5.

8.5.2 Distributions

Note that the `Delay` and `Effort` methods return a function of type `TDist`. Many of the more frequently used distribution functions are already defined, so users will rarely have to define new distribution functions. In fact, a number of standard distributions are given in Figure 8.6. The following example illustrates their use. If we want `MyAction` to have a delay that is exponentially distributed with rate 10, we can give the following definition:

```
function  TMyAction.Delay : TDist := Exponential(10);
```

8.5.3 Action partition groups

Action partition groups (see Section 3.3.3) are supported in DEF. They are constructed using a simple data structure: a linked list of linked lists. Each linked list represents partition of the actions, and the list of linked lists makes up the action partition groups. If an action is not present in this data structure, it is assumed to exist in a singular partition group.

Two DEF objects are used to represent the action partition group: `ActionGroup` and `ActionNode`. We define the class `ActionNode` first:

```
class ActionNode
   AAction a;       protected;
   ActionNode next; protected;
   procedure AddNode(ActionNode n);
end;
```

The `ActionNode` constructs a linked list with references to the action in the partition group and the next node in the linked list. The procedure `AddNode` simply adds a node to the linked list. Each linked list represents an action partition group.

The object `ActionGroup` is also a linked list:

```
TYPE
   TDist = function(t : real) : real;
   TPolicy = (DDD, DDP, DPD, DPP, PDD, PDP, PPD, PPP);

   AAction = abstract class
      constructor Create;
      function  Enabled     : boolean; abstract; pure;
      function  Delay       : TDist;   abstract; pure;
      function  Effort      : TDist;   abstract; pure;
      function  Rank        : integer; abstract; pure;
      function  Weight      : real;    abstract; pure;
      procedure Complete;              abstract;
      function  Interrupt   : boolean; abstract; pure;
      function  Policy      : TPolicy; abstract; pure;
   end;

   Action = abstract class (AAction)
      function  Enabled     : boolean; abstract; pure;
      function  Delay       : TDist;   abstract; pure;
      function  Effort      : TDist;   override; pure;
      function  Rank        : integer  override; pure;
      function  Weight      : real;    override; pure;
      procedure Complete;              abstract;
      function  Interrupt   : boolean; override; pure;
      function  Policy      : TPolicy; override; pure;
   end;

constructor AAction.Create;      begin (* Empty *) end;
function    Action.Effort      : TDist   := Zero;
function    Action.Rank        : integer := 0;
function    Action.Weight      : real    := 1;
procedure   Action.Trans;        begin (* Empty *) end;
function    Action.Interrupt   : boolean := false;
function    Action.Policy      : TPolicy := DDD;
```

Figure 8.4: Action declarations and definitions.

```
TYPE ZAction = abstract class (AAction);
      function  Enabled     : boolean; abstract; pure;
      function  Delay       : TDist;   override; pure;
      function  Effort      : TDist;   override; pure;
      function  Rank        : integer; override; pure;
      function  Weight      : real;    abstract; pure;
      procedure Complete;              abstract;
      function  Interrupt   : boolean; override; pure;
      function  Policy      : TPolicy; override; pure;
   end;

constructor ZAction.Create;      begin (* Empty *) end;
function    ZAction.Delay      : TDist   := Zero;
function    ZAction.Effort     : TDist   := Zero;
function    ZAction.Rank       : integer := 0;
function    ZAction.Interrupt  : boolean := false;
function    ZAction.Policy     : TPolicy := DDD;
```

Figure 8.5: Default zero-timed action.

```
TYPE
   function Exponential(lambda : real)       : TDist;
   function Deterministic(value : real)      : TDist;
   function Geometric(p : real)              : TDist;
   function Weibull(alpha, beta : real)      : TDist;
   function Normal(mean, var : real)         : TDist;
   function LogNormal(mu, a_sqr : real)      : TDist;
   function Erlang(m, beta : real)           : TDist;
   function Gamma(alpha, beta : real)        : TDist;
   function Beta(alpha, beta : real)         : TDist;
   function Uniform(lb, ub : real)           : TDist;
   function Binomial(time, p : real)         : TDist;
   function Hyperexp(rate1, rate2, p : real) : TDist;
   function Zero                             : TDist;
```

Figure 8.6: DEF standard distributions.

```
class ActionGroup
   ActionNode  first; protected;
   ActionGroup next;  protected;
   procedure AddGroup(ActionNode n);
end;
```

The member item `first` is a reference to the first element of a linked list that represents an action partition group. The `next` member is a link to the next action partition group. The procedure `AddGroup` adds a new partition group to the linked list.

Note that all the members are protected so that after initialization, this data structure can not be modified. This is another illustration of how small modifications to the language are necessary and convenient for model specification.

The creation of action partition groups must be done in the initialization section. One requirement not stated by the declaration is that an action must exist in at most one partition group. An error occurs if an action is placed in more than one partition group. Furthermore, actions must be placed in groups with a probability of one.

8.5.4 A simple example

We allow users to declare an object to override all the abstract members of the base class using the `override` modifier.

```
VAR MyAction : override class TMyAction (Action);
```

This automatically creates a new type called `TMyAction`. The above declaration is equivalent to declaring the following:

```
TYPE MyAction : class (Action)
      function  Enabled   : boolean; override; pure;
      function  Delay     : TDist;   override; pure;
      procedure Complete;            override;
   end;
VAR MyAction: class TMyAction;
```

We can declare a two-state Markov chain using a single object as shown in Figure 8.7. This object can be used for traffic modulation or other purposes that require a high variance. The `override` modifier eliminates the need to declare the `Enabled`, `Delay`, and `Complete` methods. The rates are declared `protected`, so once the object is initialized, the rates cannot be changed.

```
TYPE
   Tmmpp : override class(Action)
      state : boolean;
      rate1, rate2 : real; protected;
      constructor GiveRates(r1, r2 : real);
   end;
VAR
   mmpp  : Tmmpp;

constructor GiveRates(r1, r2 : real);
begin
  rate1 := r1;
  rate2 := r2;
end;

function  trans.Enabled : boolean := true;

function  trans.Delay : TDist;

begin
  if (NOT state)
    trans.Delay := Exponential(rate1);
  else
    trans.Delay := Exponential(rate2);
end;

procedure trans.Complete;
begin
   state := NOT state;
end;

to begin do begin
  mmpp.GiveRates(4,5);
end;
```

Figure 8.7: Simple, complete two-state model in DEF.

While the creation of a two-state Markov chain is much more complicated in DEF than in, for example, an SPN, there are a number of advantages to using DEF. First, objects can easily be re-used in other projects. While it may be trivial to re-draw an SPN model, it may not be trivial to redraw larger modules. Furthermore, it would not be easy to implement an array of two-state Markov chains as an SPN. Also, the object `mmpp` is given a meaningful name (for Markov-modulated Poisson process). The user can forget how the object `mmpp` is implemented. (Admittedly, for this example, the advantage is minimal.)

8.6 Measures in DEF

8.6.1 Definitions

Reward variables are constructed in DEF much as they are in the framework, with a few simplifications. We begin by illustrating the definition of the reward automata below:

```
TYPE
   TAutomata = abstract class
      procedure delta(a : AAction); abstract; local;
   end;
```

As an abstract class, no data members are declared. It is assumed that derived classes will declare data members and represent the state of the automata. The parameter to `delta` is the action that completed, and the function may read the state of the system and modify only the data members. The directive `local` ensures this syntactically. Note that `delta` is not strictly a function of an event. `delta` does not have access to the previous state or the sojourn time. Lack of access to the sojourn time ensures the Markov property.

The reward structure is defined as a class that is later used to derive the reward variable class. Class `TRS` is defined as follows:

```
   TYPE  TRS = abstract class
      function C(a : AAction) : real; abstract; pure;
      function R              : real; abstract; pure;
   end;
```

The function `C` implements the impulse reward $\mathcal{C}$, while `R` implements the rate reward $\mathcal{R}$.

The next step is the construction of the type $\mathcal{B}_{\mathrm{f}}(\mathbb{R})$. We do this by declaring a linked list of intervals, each of which has a lower bound, an upper bound, and Boolean variables that indicate whether the lower and upper bounds are closed. A number of methods accompany this class; we omit them for brevity.

```
   TBfR = abstract class
      constructor Interval(l, u : real);
      lb, ub   : real;    abstract; protected;
      lbc, ubc : boolean; abstract; protected;
      next     : TBfR;              protected;
   end;

constructor TBfR.Interval(l, u : real) begin
   lb := l;
   ub := u;
   lbc := true;
   ubc := true;
   next := nil;
end;
```

The reward control is packaged into an abstract class, which will also be used to derive the reward variable class, as Figure 8.8 illustrates. Note that functions of events in Möbius are simply functions of the actions that complete and the resulting next states. This ensures that the Markov property holds on all DEF reward variables.

Reward variable measures are also defined as an abstract class `TRMeas`, which we will use to define reward variables. Figure 8.9 shows this class definition. Only the first 10 moments can be computed (although we can easily expand this limit if we need to). Also, at most one interval is computed. If `Interval` is `Nil`, then no interval is computed.

Finally, the abstract class of a reward variable is defined:

```
TYPE
   ARV = abstract class (TAutomata, TRS, TRC, TRMeas)
   end;
```

A default reward variable definition is provided. This default can be used for "simple" reward variables that have no automata definitions and for which the utility time is fixed. The declaration and definitions are given in Figure 8.10. The derived type must provide 1) the type of the reward variable, 2) the utility time, 3) all abstract methods, and 4) the reward variable measure.

```
TYPE
   TRVType      = (Instant, Interval, TAInterval);
   TRVUTypes    = (Fixed, Variable);
   TRVTimeTypes = (Timed, Event);

   TRC = abstract class
      RVType                          : TRVType;                protected;
      function UtilityType            : TRVUTypes;    abstract; pure;
      UtilityTime                     : TBfR;                   protected;
      function StartType              : TRVTimeTypes; abstract; pure;
      function StartTime              : real;         abstract; pure;
      function StartEvent(a : AAction) : boolean;     abstract; pure;
      function Closed                 : boolean;      abstract; pure;
      function EndType                : TRVTimeTypes; abstract; pure;
      function EndTime                : real;         abstract; pure;
      function EndEvent(a : AAction)  : boolean;      abstract; pure;
      function EndUtility             : boolean;      abstract; pure;
   end;
```

Figure 8.8: Reward control in DEF.

```
TYPE
   TMoments       = set of 1..10;
   TRMeas = abstract class
      Distribution : boolean;  protected;
      Moments      : TMoments; protected;
      Interval     : TBfR;     protected;
   end;
```

Figure 8.9: Reward variable measures in DEF.

8.7 Example

We now present a complete DEF model, including the model description with reward variables. We describe a model of a very simple system, because we are trying to model a useful, realistic system, but rather to illustrate the usefulness of the DEF modeling language.

Consider a simple multiprocessor system in which all processors service tasks from a single, degradable buffer. The normal, fault-free operation of the system is as follows. Tasks arrive as a Poisson process with rate `ALPHA`. If the buffer is full, then the process is rejected. Otherwise, it is placed in the buffer to be served by the first available processor in a FIFO manner. The processing time for each process is independent and exponentially distributed

```
TYPE
   SRV = abstract class (ARV)
      procedure delta(a : AAction);              override; local;
      function  C(a : AAction) : real;           abstract; pure;
      function  R              : real;           abstract; pure;
      function  UtilityType    : TRVUTypes;      override; pure;
      function  StartType      : TRVTimeTypes;   override; pure;
      function  StartTime      : real;           override; pure;
      function  StartEvent(a : AAction):boolean; override; pure;
      function  Closed         : boolean;        override; pure;
      function  EndType        : TRVTimeTypes;   override; pure;
      function  EndTime        : real;           override; pure;
      function  EndEvent(a : AAction) : boolean; override; pure;
      function  EndUtility     : boolean;        override; pure;
   end;

procedure SRV.delta(a : AAction); begin (* do nothing *) end;
function  UtilityType : TVUTypes := fixed;
(* All functions below are arbitrary because UtilityType fixed *)
function  SRV.StartType : TRVTimeTypes := Fixed;
function  SRV.StartTime : TRVTime := 0;
function  SRV.StartEvent(a : AAction) : boolean := false;
function  SRV.Closed : boolean := false;
function  SRV.EndType : TRVTimeTypes := Fixed;
function  SRV.EndTime : real := 0;
function  SRV.EndEvent(a : AAction) : boolean := false;
function  SRV.EndUtility : boolean := false;
```

Figure 8.10: "Simple" reward variable declaration and definition.

with a processing rate `BETA`.

Faults can occur because of failure of either a buffer stage or a processor. In either case, the fault may either be covered (i.e., the system degrades successfully), or result in a total loss of processing capability (i.e., total system failure). Furthermore, certain processor failures are repairable. Repairs are performed on one processor at a time, with exponentially distributed repair time with rate `ZETA`. We assume further that faults in both a buffer stage and a processor arrive as a Poisson process with rates `LAMBDA` and `GAMMA` respectively. The following is a DEF listing that implements the described model:

```
CONST
   ALPHA  = . . .;        (* fill in appropriate values *)
   BETA   = . . .;
```

```
   LAMBDA = . . .;
   GAMMA  = . . .;
   ZETA   = . . .;

   NUM_PROCS = 3;        (* Various system parameters *)
   NUM_BUFFS = 2;

   PROB_PROC_REPAIRABLE = . . .;
   PROB_PROC_COVERED    = . . .;
   PROB_PROC_UNCOVERED  = . . .;
   PROB_BUFF_COVERED    = . . .;
   PROB_BUFF_UNCOVERED  = . . .;

VAR
   Procs, Repairs : [0 .. NUM_PROCS];
   Buffs          : [0 .. NUM_BUFFS];
   ProcFailed,
   BuffFailed     : boolean;

   ProcFails        : class A_ProcFails (Action);       override;
   ProcRepairable   : class A_ProcRepairable(ZAction);  override;
   ProcCovered      : class A_ProcCovered(ZAction);     override;
   ProcUncovered    : class A_ProcUncovered(ZAction);   override;

   ProcRepair       : class A_ProcRepair(Action);       override;

   BufferFails      : class A_BufferFails(Action);      override;
   BufferCovered    : class A_BufferCovered(ZAction);   override;
   BufferUncovered  : class A_BufferUncovered(ZAction); override;

(*
 * Various action definitions
 *)

function A_ProcFails.Enabled : boolean := Procs > 0;
function A_ProcFails.Delay : real := Exponential(GAMMA * Procs);
procedure A_ProcFails.Complete; begin
   Procs := Procs - 1;
   ProcFailed := true;
end;

function A_ProcRepairable.Enabled : boolean := ProcFailed;
function A_ProcRepairable.Weight : real := PROB_PROC_REPAIRABLE;
procedure A_ProcRepairable.Complete; begin
   ProcFailed := false;
```

```
   Repairs := Repairs + 1;
end;

function A_ProcCovered.Enabled : boolean := ProcFailed;
function A_ProcCovered.Weight : real := PROB_PROC_COVERED;
procedure A_ProcCovered.Complete; begin
   ProcFailed := false;
end;

function A_ProcUncovered.Enabled : boolean := ProcFailed;
function A_ProcUncovered.Weight : real := PROB_PROC_UNCOVERED;
procedure A_ProcUncovered.Complete; begin
   Procs := 0;
   ProcFailed := false;
   Buffs := 0;
   Repairs := 0;
end;

function A_ProcRepair.Enabled : boolean := Repairs > 0;
function A_ProcRepair.Delay : TDist := Exponential(ZETA);
procedure A_ProcRepair.Complete; begin
   Repairs := Repairs - 1;
   Procs := Procs + 1;
end;

function A_BufferFails.Enabled : boolean := Buffs > 0;
function A_BufferFails.Delay : TDist := Exponential(LAMBDA * Buffs);
procedure A_BufferFails.Complete; begin
   Buffs := Buffs - 1;
   BuffFailed := true;

function A_BufferCovered.Enabled : boolean := BuffFailed = 1;
function A_BufferCovered.Weight : real := PROB_BUFF_COVERED;
procedure A_BufferCovered.Complete; begin
   BuffFailed := false;
end;

function A_BufferUncovered.Enabled : boolean := BuffFailed;
function A_BufferUncovered.Weight : real := PROB_BUFF_UNCOVERED;
procedure A_BufferCovered.Complete; begin
   Procs := 0;
   Buffs := 0;
   BuffFailed := false;
   Repairs := 0;
end;
```

```
(* Set initial state *)
to begin do
   Procs := NUM_PROCS;
   Buffs := NUM_BUFFS;

   (* Create all the actions *)
   ProcFails.Create;
   ProcRepairable.Create;
   ProcCovered.Create;
   ProcUncovered.Create;
   ProcRepair.Create;
   BufferFails.Create;
   BufferCovered.Create;
   BufferUncovered.Create;
end;
```

This defines the failure behavior of the system as an atomic model expressed in DEF. We can define reward variable measures on this model. For the reliability measure, the system is said to be operational if there is at least one buffer and there is at least one processor that is either operating properly or repairable. For the availability measure, the system is said to be operational if at least one processor is operating properly.

The following DEF listing contains a number of reward variables. The reward variable `Reliability` computes the reliability at time `TIME` (an arbitrary value for this example). `Availability` computes the expected instant-of-time availability at time `TIME`. These two reward variables use only rate rewards for their computation. `NumRepairs` uses impulse rewards to count the number of repairs in the interval $[0, \texttt{TIME}]$. All of these reward variables are based on the "simple" reward variable type `SRV`.

The reward variable `MTTF` measures the time until the system fails. It does this by having a constant rate reward of 1, and a utility time that starts at time zero and ends on the first event in which the system enters a failed state. The (global) function `SystemFailed` returns `true` whenever the system has failed. Since all the reward variables can use the same function for determining whether the system has failed, the definition can be changed easily without modification of any of the reward variables.

To illustrate the use of automata, we create an automaton that measures whether three consecutive processor repairs take place before a total system failure. We use the reward variable `ThreeRepairs` for this. The automaton advances from `NoR` (no consecutive repairs) to `OneR` (one consecutive repair) if a repair takes place. When the automaton is in `OneR`, if a covered, unrepairable processor failure occurs, the automaton resets; if another repair

occurs, then the automaton is advanced to `TwoR` (two consecutive repairs); if anything else occurs, the automaton remains in the current state. A similar transition scheme exists for the state `TwoR`. When the automaton reaches `ThreeR`, it halts. The utility time is an instant of time that is determined by the model. The instant occurs when the system fails or three consecutive repairs occur. The value of the reward variable is 1 if the automata is in state `ThreeR` at the utility time instant.

Solvers can benefit from reward variables described in that way. Since the solvers need perform computation only for the utility time, it can help reduce unnecessary computation if the solver can determine the smallest possible utility time (in the case of `ThreeRepairs`, for example):

```
CONST
   TIME = . . .;

TYPE
   Reliability  : override class (SRV);
   Availability : override class (SRV);
   NumRepairs   : override class (SRV);
   MTTF         : override class (ARV);
   ThreeRepairs : override class (ARV)
      automata : (NoR, OneR, TwoR, ThreeR); (* Enumerated states *)
   end;

VAR
   ival : TBfR;
   inst : TBfR;

function SystemFailed : boolean :=
   (Procs + Repairs = 0 AND NOT ProcFailed) OR
         (Buffs = 0 AND NOT BuffFailed);

constructor Reliability.Create; override; begin
   RVType := Interval;
   UtilityTime := ival;
   Distribution := false;
   Moments := [1];
   Intervalse := nil;
end;
function Reliability.C(a : AAction) : real := 0;
function Reliability.R : real;
   if SystemFailed then Reliability.R := 1; else Reliability.R := 0;
end;
```

```
constructor Availability.Create; override; begin
   RVType := Instant;
   UtilityTime := inst;
   Distribution := false;
   Moments := [1];
   Intervals := nil;
end;
function Availability.C(a : AAction) : real := 0;
function Availability.R : real := Procs > 0 AND Buffs > 0;

constructor NumRepairs.Create begin
   RVType := Interval;
   UtilityTime := ival;
   Distribution := true;
   Moments := [0];
   Intervals := nil;
end;
function NumRepairs.C(a : AAction) : real := a = ProcRepair;
function NumRepairs.R : real := 0;

constructor MTTF.Create; override; begin
   RVType := Interval;
   UtilityType := Variable;
   UtilityTime := nil;
   Distribution := true;
   Moments := [1,2];
   Intervals := ival;
end;
procedure MTTF.delta(a : AAction); begin end;
function  MTTF.C(a : AAction) : real := 0;
function  MTTF.R : real := 1;
function  MTTF.UtilityType : TRVUTypes := Variable;
function  MTTF.StartType : TRVTimeTypes := Timed;
function  MTTF.StartTime : real := 0;
function  MTTF.StartEvent(a : AAction) : boolean := false; (* arbitrary *)
function  MTTF.Closed : boolean := true;
function  MTTF.EndType : TRVTimeTypes := Event;
function  MTTF.EndTime : real := 0; (* arbitrary *);
function  MTTF.EndEvent(a : AAction) : boolean := SystemFailed;
function  MTTF.EndUtility : boolean := true;

constructor ThreeRepairs.Create; override; begin
   RVType : Instant;
   UtilityType := Variable
   UtilityTime := nil;
```

```
   Distribution := true;
   Moments := [1];
   Intervals := nil;
end;
procedure ThreeRepairs.delta(a : AAction); begin
   case automata of
      NoR:
         if (a = ProcRepair) then automata := OneR;
      OneR:
         if (a = ProcRepair) then
            automata := TwoR;
         else if (a = ProcCovered) then
            automata := NoR;
      TwoR:
         if (a = ProcRepair) then
            automata := ThreeR;
         else if (a = ProcCovered) then
            automata := NoR;
      ThreeR: (* Do nothing *)
   end;
end;
function ThreeRepairs.C(a : AAction) : real := 0;
function ThreeRepairs.R : real := automata = ThreeR;
function ThreeRepairs.UtilityType : TRVUTypes := Variable;
function ThreeRepairs.StartType : TRVTimeTypes := Event;
function ThreeRepairs.StartTime : real := 0;
function ThreeRepairs.StartEvent(a : AAction) : boolean; begin
   if SystemFailed then ThreeRepairs.StartEvent := true;
   if automata = ThreeR then ThreeRepairs.StartEvent := true;
   ThreeRepairs.StartEvent := false;
end;
(* These are arbitrary because RV is instant-of-time *)
function ThreeRepairs.Closed : boolean := true;
function ThreeRepairs.EndType : TRVTimeTypes := Timed
function ThreeRepairs.EndTime : real := 0;
function ThreeRepairs.EndEvent(a : AAction) : bool := false;
(* Not arbitrary *)
function ThreeRepairs.EndUtility : boolean := true;

to begin do
   ival.Interval(0, TIME);
   inst.Interval(TIME,TIME);

   (* Create reward variables *)
   Reliability.Create;
```

```
   Availability.Create;
   NumRepairs.Create;
   MTTF.Create;
   ThreeRepairs.Create;
end;
```

8.8 Conclusion

We argue that DEF has many of the features that are important to any good modeling formalism. We believe that the quality of a modeling formalism can be judged on a number of criteria. We list some of them here.

- Models should be easy to build.
- Models should be easy to understand, communicate, and document. The formalism should be easy to learn and intuitive.
- The formalism should be scalable; it should be able to manage complexity as the model grows in detail.
- A modeling formalism should be expressive enough that modelers can build models that represent systems from a wide range of system domains.
- It should be easy, or at least possible, to validate models.
- Models should afford efficient solution when possible.

DEF is a modeling language that is more powerful and expressive than any extension of stochastic Petri nets, and nearly as powerful as many simulation languages. Unlike simulation languages, DEF allows Markov-based solutions. Thus, DEF fits the niche between simple analytic models and complex simulation models. Since simple Markov models as well as complicated system structure can be expressed in DEF, it is an ideal language for models that increase in complexity as the project evolves. This is important because models can be archived, refined, compared, and extended using all the features of object-oriented programming, without the need to use multiple tools or formalisms.

Models are easy to build in DEF because it is an object-oriented language, and portions of a model can be readily reused. State and the procedures and functions that operate on the state are packaged together. Libraries of components can be built, used, and modified using well-established object-oriented techniques.

Models should be easy to understand, communicate, and document. DEF accomplishes this in a number of ways. First, DEF is based on Pascal, a well-known programming language. Thus, DEF will be relatively easy to learn, and it will also be easy for users to understand and communicate models, as DEF uses expressions and statements in a form that is generally familiar (unlike, for example, Petri nets). With DEF, a user is able to give meaningful names to every state variable, object, procedure, function, and type.

DEF is entirely textual. Sometimes this makes it more difficult to recognize simple behaviors than it would be in graphical formalisms. However, it also means that more complex behaviors can be described succinctly. In SRNs or SANs, complex behaviors are expressed using special functions, which hide the complex behavior, or gates, which contain a programming-language-based description of the complex behavior. DEF is consistent in using only one format (textual, programming-language-like) for describing a model.

As it is based on Pascal, DEF is able to declare virtually any data structure to be a state variable. Sets, pointers, enumerated types, arrays, and records are basic types. The use of data structures greatly enhances the readability and usability of models. Other formalisms, specifically those based on Petri nets, are restricted in that they may only use integer data types. This sometimes requires modelers to force unusual and unnatural encodings of state into integers. Although that is possible, it makes models more obscure.

Documentation of DEF models can be done inline with the model text in the form of comments. Comments aid in the readability and communicability of models. A variety of software engineering tools that deal with comments and objects can be easily adapted for use in DEF.

Finally, solution should be efficient. DEF offers simulation as a solution method; for models with exponential delays, it offers Markov-based solution methods as well. The use of special data types, such as subranges and enumerated types, is especially useful in state-based methods, because special data types can reduce the size of a state representation to a minimum. Furthermore, if objects are used, the most space-efficient or time-efficient data structures can be implemented and easily substituted, depending on whether space or time efficiency is of primary concern.

In all of the ways described, DEF shows itself to be a formalism with many unique abilities not found in existing formalisms.

CHAPTER 9

CONCLUSION

9.1 Review

9.1.1 Introduction

We began the thesis by describing the Möbius framework. Möbius is motivated by the desire to integrate many modeling techniques into a single tool. Since no single formalism or solution technique is always best, we argue that a framework that includes multiple modeling formalisms and solvers, meets the needs of users and developers more readily than a single technique would. We described various tools, and explain how Möbius is different in that it integrates the various modeling techniques more tightly than existing tools. We described Möbius first through a high-level overview, including various model components.

9.1.2 A flexible execution policy

We described the Möbius execution policy at a conceptual level in Chapter 2. The Möbius execution policy is different from existing policies in that it needs to be able to capture the behavior of the various formalism execution policies, yet retain sufficient structure to facilitate efficient solution. We described the more well-used execution policies, and some of the fundamental differences between them. We then posed an example that could not be represented except by the most general simulation languages, in order to further explain the need for a more general execution policy. This illustrated the limitations of the "constant work" assumption used by other execution policies.

We argued that many formalism execution policies are limited in a way that affords efficient analytic solution. The Möbius framework is concerned with the ability to express behaviors while facilitating efficient solution, and is not as focused on the solution itself. We then described the Möbius execution policy, which fulfills all of the needs we expressed,

including relaxation of the "constant work" assumption. We developed the Möbius execution policy which is able to capture all of the behaviors that the various formalism execution policies are able to express, and more. The price for this generalization is that the action state requires five variables, including three real-valued variables, whereas other execution policies often use only two. However, other execution policies are limited in that they can capture only one of two types of behavior at a time; Möbius is able to capture both at the same time. We illustrated how the Möbius execution policy is able to solve the example we posed earlier. We concluded with a discussion that shows that a solution by simulation of a model using the Möbius execution policy can be more computationally expensive than traditional policies, but the extra cost need only be incurred when using the unique features of Möbius. Thus, simulating models of existing formalisms can be done efficiently.

9.1.3 Atomic models

In Chapter 3, we addressed atomic models, the most basic Möbius model type. The Möbius atomic model is made up of state variables, actions, and properties. State variables represent various configurations that the system may be in. State variables have types and values. The type can be any one of a rich set of types, including integers, reals, pointers, arrays, and unions. The values that a state variable can take are determined by the type of that state variable. An atomic model is also provided with a set of initial states and the probability distribution over those states. State variables may also have properties, which are symbols with proprietary meaning. Properties are used by formalisms to convey special, proprietary information about a model to the solvers, so that the solvers may take advantage of that information for efficient solution.

Actions change the state of a model over time. Actions are made up of a number of action functions, which indicate, for example, when the action is enabled, what happens when an action completes or becomes disabled, and priorities among actions that are scheduled to complete at the same time. An action state, which is used to describe the state of the model, includes the five variables needed to support the Möbius execution policy. Each actions also includes an action partition group, which is used as part of the resolution algorithm that determines what to do when multiple actions are scheduled to complete at the same time. Action weights are only used to compute the relative probability of completion of actions within the same action partition group. If actions of different action partition groups are competing, the well-specified algorithm is used to determine whether measurable ambiguity exists. Chapter 6 discusses the well-specified check.

We then formally defined the atomic model as a collection of state variables, actions, and

model properties. We illustrated how SRNs can be expressed in the Möbius framework by formally showing a mapping from an SRN model to a Möbius model.

9.1.4 Möbius reward variables

We began by discussing reward variables used by several formalisms. We chose three representative reward variable formalisms: SANs, path-based, and SRN reward variables. We described each of these in detail. We also noted other techniques similar to reward variables, namely CSL and CTL, which are not supported by Möbius, and we discussed why they are not supported.

We described the shortfalls of each of the formalism reward variables. While SRN reward variables are expressed very generally, they lack any structure by which a tool can efficiently calculate them. Furthermore, the utility time is assumed to be $[0, \infty)$, which precludes efficient solution. SANs have more structure, but lack the ability to measure sequences of events. Path-based reward variables were designed to address that shortfall, but are limited in other ways, most notably in the utility time. Neither SAN nor path-based reward variables include any information about the nature of the solution, i.e., whether to solve in distribution or only the mean.

Möbius takes a bold and different approach to reward variables. All the information about a reward variable is encapsulated within the reward variable definition. This includes the automata, the utility time, and the reward variable measure. The automata, like with path-based reward variables, is used to measure sequences of events. The utility time is allowed to be fixed, or to depend on events within the model. The reward variable measure informs the solvers what the user wants to know about the solution, i.e., what characterization or measure of the reward variable is desired. Thus, we are able to construct a very general reward variable definition without sacrificing structure.

We described the algorithm used by reward variables for collecting reward. We then showed formally how to compute the reward variables on a Möbius model.

We compared the Möbius reward variables to the representative formalism reward variables, and showed that Möbius reward variables are able to capture all the behaviors of the formalisms and more.

9.1.5 Formal Möbius execution policy

Chapter 5 formally described the Möbius execution policy. It was described inductively. The model gives (probabilistically) the initial state of the model, and the execution policy

describes how the model evolves from the initial state.

We define two stochastic processes: the "stochastic process," which is used for reward variables and other purposes, and the "augmented stochastic process," which is used for describing the execution policy. The reason for defining two stochastic processes is that there are two different views of a model execution. One view is that a model execution can be represented by a sequence of events. This representation is sufficient for defining such things as reward variable automata, reward variable utility times, and for triggering interrupting events. While it is sufficient for describing the future behavior of the model, it is inconvenient for that purpose. The augmented stochastic process is defined to fill that need.

The augmented stochastic process is a sequence of augmented events. An augmented event is made up of the state of the state variables, actions, and reward variable automata, the sojourn time, and the action that completes.

Given the augmented state ς_i, we show how to determine the sojourn time and the action that completes. Then, given the sojourn time and action that completes, we show how to compute the next augmented state ς_{i+1}. This description is formal, and we give both an English description and an algorithmic description.

9.1.6 An efficient well-specified checker

Sometimes the model description is incomplete. Möbius allows for the stochastic behavior to be ambiguous as long as the ambiguity is resolved in zero time and does not affect the values of any reward variables. Thus, the ambiguity may not be timed or measurable. In Chapter 6 we discussed two well-known approaches for dealing with this level of ambiguity and determining whether the ambiguity is acceptable. These are the well-specified check, which has been applied only to SANs, and the well-defined check, which has been applied only to SPNs.

The existence of the two checks poses a problem for Möbius, because two different criteria are used for two different formalisms. If both formalisms exist in the same model, solvers must choose to apply one broadly, or apply the two selectively, which is further complicated if the models created in different formalisms interact in ambiguous ways. Furthermore, the well-specified check is extremely costly in terms of computational resources, while the well-defined check is relatively inexpensive.

We show by proof that the two checks, for all practical purposes, perform the same check. The only difference between the two checks involves sample paths that may occur with probability zero, which are not of any engineering interest. Thus, we solve two problems. First,

the task of solvers is significantly simplified in that only one check need be applied. Second, we determined that the computationally inexpensive well-defined check can adequately perform the well-specified check. We concluded with a description of the algorithm.

9.1.7 Other model types

We concluded our formal description of the Möbius framework in Cahpter 7 with several remaining model types. First, we described composed models, which are models that are typically formed by taking several reward models and modifying them slightly so that the models become structurally joined, either by sharing state or by synchronizing on events. Properties require special consideration in the composition process.

Next, we described the parameterized model. A parameterized model is augmented with parameters, which are simply variables with real-numbered values. Most of the components of a model may be functions of parameter values. For example, a failure rate may be set by a model parameter. Parameterized models are used in study models and connected models, described below.

A solved model is a reward model with results. Results are the computed solution to the reward variable measures of the reward variables with any additional information in the form of annotations. Annotations are a way of recording any solver-specific information, such as the confidence interval, the number of iterations, or a maximum error range. A solver maps a reward model to a solved model.

A connected model is a collection of reward models that share results. The results of one model can change the parameter values of other models. Thus, an overall connected model solution can be computed by solving the submodels, possibly iteratively, until some stopping criterion is met. Model connection is a technique that can be used to drastically reduce solution time in some models. Connected models are used, for example, in decompositional techniques.

A study model is a parameterized model with a set of parameter values. Solvers should solve the reward model for all the parameter values. Thus, a range of parameter values can be conveniently solved for. While not of much scientific interest, study models are convenient for the exploration of the design space of a system. We also described the connected study model, which allows for study of a connected model.

9.1.8 DEF example formalism

In Chapter 8, we describe a novel formalism, called DEF, developed specially for Möbius. DEF has many features of an object-oriented programming language that are found in many modern simulation languages. It is a unique hybrid of high-level modeling languages and automaton-based formalisms such as stochastic Petri nets and extensions.

DEF is able to capture most of the features of the Möbius atomic and reward models. This is because we developed a predefined set of objects that represent actions and reward variables. These objects closely resemble the formal specifications. Although designed to illustrate the features of Möbius, DEF is in itself a powerful and useful modeling formalism.

DEF has many advantages over existing formalisms. First, it is more expressive and flexible than any of the extensions of stochastic Petri nets. That allows more complex behaviors and structures to be easily represented in a model. DEF is nearly as expressive as many simulation languages. Unlike simulation languages, DEF can also facilitate a Markov chain-based solution if the delays are exponential or zero-timed. Thus, DEF offers the advantages of the expressiveness of simulation languages, but with the accuracy and speed of the Markov chain-based solutions afforded by stochastic Petri nets.

9.1.9 Putting it all together

The volume of notation can be daunting. We include here Figure 9.1 which outlines the various formal Möbius components for the reader's convenience. The figure shows the structure of the basic framework components, and the hierarchical relationship between them. It illustrates how the model types are built up out of other model types and components. This figure is the formal complement to Figure 1.1. Figure 1.1 shows a high-level diagram of the various framework components, and Figure 9.1 shows the same information using the formal notation.

9.2 Prototype Software Environment

The utility of the Möbius framework has been demonstrated by the implementation of atomic model formalisms, composed model formalisms, and a reward variable formalism, as well as solvers [83], [84], within a software environment. The software environment that we have developed permits the inclusion of new model formalism editors and model solution engines as they are developed. Within this environment, we have implemented a number of atomic model editors [85] based on stochastic activity networks [56], [77], Buckets and Balls

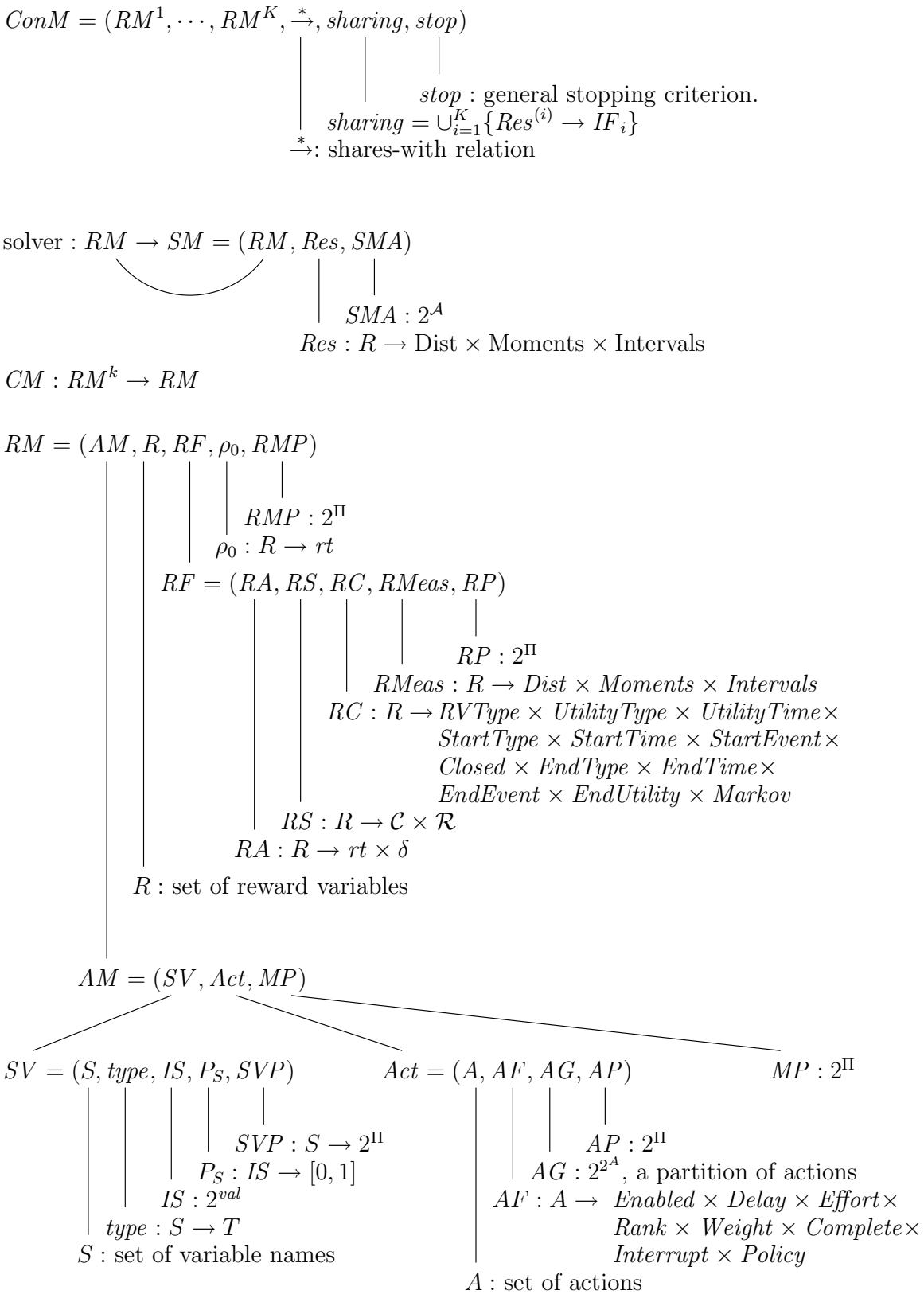

Figure 9.1: Framework component diagram.

[86], and PEPA [37]. Composed model editors include the Replicate/Join editor [87] which implements the replicate/join model composition language [36], and a composed model editor which implements a graph composition formalism of [42]. The tool also implements a rate/impulse reward specification language based on the SAN-based reward variable specification language [45].

The Möbius tool also implemented two solution techniques: a simulation engine [88] that implemented both terminating and steady-state simulation (via batch means), and a state-space generator with various analytic/numerical solvers [89]. The analytic/numerical solvers provided for transient instant-of-time and interval-of-time measures, as well as steady-state measures.

While some of these modeling languages and techniques have been previously implemented in the modeling tool *UltraSAN* [15], their reimplementation in the Möbius framework showed that the base classes and the abstract functional interface can be used to hide modeling-language-specific details from the models that interact with one another and with the model solution techniques.

Moreover, the act of implementing multiple nontrivial model specification languages using the Möbius abstract functional interface helps us to precisely define the base classes themselves, by helping us determine how multiple modeling languages should best interact with one another when they have minimal knowledge of each other's internal representations. Finally, the implementation allowed us to find, through experimentation, an implementation of the base classes that hides as many details as possible of individual modeling formalisms while still providing good performance in the resulting tool. In particular, Möbius was able to obtain simulation model solutions in approximately the same time as our previous (specialized) tool *UltraSAN* could. Also, we found that state-space generation could be performed slightly faster and with less memory than in *UltraSAN*. These results prove that the Möbius framework shows promise as a way to build software environments with multiple model specification methods at different levels of detail and abstraction, multiple model composition and connection methods, and multiple model solution methods.

9.3 Lessons Learned

The framework and the tool have evolved throughout this experience. As is frequently the case, implementation lags behind theory. Even so, the development has been very successful so far. Even more important, the practice of building a software tool has given us valuable feedback on how to refine a formal framework. The Möbius framework is a product of that

process, and it could not be as mature a framework without it.

We also concluded that many existing formalisms are unable to show the expressiveness available in the Möbius framework. We wanted to create a new formalism that would express much, if not all of the expressiveness that is possible within an atomic modeling formalism and reward variables. That would give people an idea of the possibilities that are present in creating formalisms within the framework.

We also encountered a number of other fundamental research questions, such as what to do when there are multiple events scheduled to occur at the same time and no ordering of those events is specified. Current algorithms exist to deal with this problem, but the best solution to date is exponential in complexity. We addressed this problem.

9.4 Future Work

With the completion of the framework's formal specification, the Möbius project is still only beginning. One of its goals is to facilitate future modeling research, and Möbius provides the mechanism by which we can begin to perform that research. We need to see two "proofs" of the quality of the formal specification. First, we need to know whether a tool can be built that conforms to the formal specification and intentions of the Möbius framework. This "proof" has already been provided in the form of the Möbius tool and the number of formalisms already supported by the tool. Another "proof" will be provided if we find that new techniques can be successfully facilitated and integrated into the Möbius tool. We will consider Möbius successful if new techniques are developed that could or would not have been developed without Möbius.

While the tool has demonstrated the viability of Möbius, the unfortunate reality is that the tool and the framework were developed in parallel, as a result of which the two do not match in every detail. The tool should be modified so that it more closely matches the details and philosophy presented in the framework.

Formalism developers who which to integrate modeling formalisms into the Möbius framework, particularly atomic and reward formalisms, should probably perform a formal translation, as we did for SRNs in Section 3.5.

Finally, Möbius was designed to be extensible. As new challenges arise, the formal specification will undoubtedly need to be modified in order to meet these challenges. We designed Möbius with the hope that modifications would not be necessary, especially to the atomic and reward variable models. However, we do anticipate that it may be necessary to add new model types, like the study model or the connected model. These new model types

will not change the fundamental components of the framework; they rather will build on them.

REFERENCES

[1] 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.

[2] H. Beilner, J. Mäter, and N.Weißenberg, "Towards a performance modelling environment: News on HIT," in *Proceedings of the 4th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, 1988, pp. 57–75.

[3] R. G. Franks, A. Hubbard, S. Majumdar, J. E. Neilson, D. C. Petriu, J. Rolia, and C. M. Woodside, "A toolset for performance engineering and software design of client-sever systems," *Performance Evaluation*, vol. 24, no. 1–2, pp. 117–136, Nov. 1995.

[4] M. Veran and D. Potier, "QNAP2: A portable environment for queuing system modeling," in *Modeling Techniques and Tools for Computer Performance Evaluation*, D. Potier, Ed., 1984, pp. 25–63.

[5] K. C. Chang, R. F. Gordon, P. G. Loewner, and E. A. MacNair, "The RESearch Queueing package Modeling Environment (RESQME)," IBM, Yorktown Heights, NY, Research Report RC-18687, Feb. 1993.

[6] S. Christensen and K. H. Mortensen, "Tools on the web," `http://www.daimi.au.dk/~petrinet/tools/`.

[7] M. Trompedeller, "A Petri net classification and related tools," `http://www.dsi.unimi.it/Users/Tesi/trompede/petri/home.html`.

[8] C. Lindemann, A. Reuys, and A. Thümmler, "The DSPNexpress 2.000 performance and dependability modeling environment," in *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing (FTCS '99)*, Madison, Wisconsin, June 1999, pp. 228–231.

[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] F. Bause, P. Buchholz, and P. Kemper, "QPN-tool for the specification and analysis of hierarchically combined queuing Petri nets," in *Quantitative Evaluation of Computing and Communication Systems: Proceedings of the 8th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (Performance Tools '95)*, H. Beilner and F. Bause, Eds., Heidelberg, Germany, Sept. 1995, Springer, pp. 224–238.

[11] R. German, "SPNL: Processes as language-oriented building blocks of stochastic Petri nets," in *Computer Performance Evaluation; Modelling Techniques and Tools; Proceedings of the 9th International Conference*, R. Marie, B. Plateau, M. Calzarossa, and G. Rubino, Eds., St. Malo, France, June 1997, pp. 123–134.

[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] C. Beounes, M. Aguera, J. Arlat, S. Bachmann, C. Bourdeau, J. E. Doucet, K. Kanoun, J. C. Lapries, S. Metge, J. Moreira de Souza, D. Powell, and P. Spiesser, "SURF-2: A program for dependability evaluation of complex hardware and software systems," in *Proceedings of the 23rd Annual International Symposium on Fault-Tolerant Computing (FTCS '23)*, Toulouse, France, June 1993, pp. 668–673.

[14] R. German, C. Kelling, A. Zimmerman, and G. Hommel, "TimeNET: A toolkit for evaluating non-Markovian stochastic Petri-nets," *Performance Evaluation*, vol. 24, pp. 69–87, 1995.

[15] 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.

[16] K. K. Goswami and R. K. Iyer, "DEPEND: A simulation-based environment for system level dependability analysis," *IEEE Transactions on Computers*, vol. 46, no. 1, pp. 60–74, Jan. 1997.

[17] M. Bouissou, "The FIGARO dependability evaluation workbench in use: Case studies for fault-tolerant computer systems," in *Proceedings of the 23rd Annual International*

Symposium on Fault-Tolerant Computing (FTCS '23), Toulouse, France, June 1993, pp. 680–685.

[18] S. J. Bavuso, J. B. Dugan, K. S. Trivedi, E. M. Rothmann, and W. E. Smith, "Analysis of typical fault-tolerant architecture using HARP," *IEEE Transactions on Reliability*, vol. 36, no. 2, pp. 176–185, 1987.

[19] M. Sridharan, S. Ramasubramanian, and A. K. Somani, "HIMAP: Architecture, features, and hierarchical modeling specification techniques," in *Computer Performance Evaluation: Modelling Techniques and Tools: Proceedings of the 10th International Conference on Tools (Tools '98)*, R. Puigjaner, N. N. Savino, and B. Serra, Eds., Palma de Mallorca, Spain, Sept. 1998, pp. 348–351.

[20] A. Goyal, W. C. Carter, E. de Souza e Silva, S. Lavenberg, and K. S. Trivedi, "A system availability estimator," in *Proceedings of the 16th Annual International Symposium on Fault-Tolerant Computing (FTCS '16)*, 1986, pp. 84–89.

[21] C. U. Smith and L. G. Williams, "Performance engineering evaluation of object-oriented systems with SPE·ED^{TM}," in *Computer Performance Evaluation; Modelling Techniques and Tools; Proceedings of the 9th International Conference*, R. Marie, B. Plateau, M. Calzarossa, and G. Rubino, Eds., St. Malo, France, June 1997, pp. 135–154.

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

[23] C. U. Smith, "Integrating new and 'used' modeling tools for performance engineering," 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. 153–163.

[24] 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.

[25] R. Fricks, S. Hunter, S. Garg, and K. Trivedi, "IDEA: Integrated design environment for assessment of ATM networks," in *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems*, Montreal, Canada, Oct. 1996, pp. 27–34.

[26] 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.

[27] A. P. A. van Moorsel and Y. 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.

[28] R. A. Sahner, "Combinatorial-Markov methods of solving performance and reliability models," Ph.D. dissertation, Duke University, Durham, North Carolina, 1986.

[29] 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*. Boston: Kluwer, 1996.

[30] G. Ciardo and A. S. Miner, "SMART: Simulation and Markovian analyzer for reliability and timing," in *Proceedings of the International Computer Performance and Dependability Symposium (IPDS '96)*, Urbana-Champaign, Illinois, Sept. 1996, p. 60.

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

[32] F. Bause and H. Beilner, Eds., "Performance tools: Model interchange formats," Fachbereichs Informatik der Universität Dortmund, Germany, Forschungsbericht (Research Report) Nr. 581, 1995.

[33] F. Bause, P. Buchholz, and P. Kemper, "A toolbox for functional and quantitative analysis of DEDS," in *Computer Performance Evaluation: Modelling Techniques and Tools: Proceedings of the 10th International Conference*, B. S. R. Puigjaner, N.N. Savino, Ed., Palma de Mallorca, Spain, Sept. 1998, pp. 356–359.

[34] F. Bause, P. Kemper, and P. Kritzinger, "Abstract Petri net notation," *Petri Net Newsletter*, vol. 49, pp. 9–27, 1995.

[35] 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.

[36] 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.

[37] G. Clark and W. H. Sanders, "Incorporating stochastic process algebra into the Möbius framework," in *Proceedings of Process Algebra and Performance Modelling (PAPM 2001)*, Aachen, Germany, Sept. 2001, pp. 200–215.

[38] 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.

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

[40] 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.

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

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

[43] D. D. Deavours and W. H. Sanders, "'On-the-fly' solution techniques for stochastic Petri nets and extensions," in *Proceedings of the Seventh International Workshop on Petri Nets and Performance Models*, Saint Malo, France, June 1997, pp. 132–141.

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

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

[46] 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, pp. 251–260.

[47] 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.

[48] 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.

[49] S. Donatelli, "Superposed generalized stochastic Petri nets: Definition and efficient solution," in *Application and Theory of Petri Nets*, , R. Valette, Ed. June 1994, pp. 258–277, Zaragoza, Spain.

[50] 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.

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

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

[53] P. W. Glynn, "A GSMP formalism for discrete event systems," *Proceedings of the IEEE*, vol. 77, no. 1, pp. 14–23, Jan. 1989.

[54] R. German, A. van Moorsel, M. A. Qureshi, and W. H. Sanders, "Algorithms for the generation of state-level representations of stochastic activity networks with general reward structures," *IEEE Transactions on Software Engineering*, vol. 22, pp. 603–614, Sept. 1996.

[55] 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.

[56] 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.

[57] 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.

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

[59] 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)*, Durham, North Carolina, Oct. 1995, pp. 113–122.

[60] R. German, *Performance Analysis of Communication Systems with Non-Markovian Stochastic Petri nets*. Chichester, England: John Wiley & Sons, 2000.

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

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

[63] 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.

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

[65] 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.

[66] G. Ciardo, A. Blakemore, P. F. J. Chimento, J. K. Muppala, and K. S. Trivedi, "Automatic generation and analysis of Markov reward models using stochastic reward nets,"

in *Linear Algebra, Markov Chains, and Queueing Models*, , C. Meyer and R. J. Plemmons, Eds. Heidelberg: Springer-Verlag, 1993, pp. 141–191.

[67] R. A. Howard, *Dynamic Probabilistic Systems, Vol. II: Semi-Markov and Decision Processes*. New York: Wiley, 1971.

[68] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen, "Model checking continuous-time Markov chains by transient analysis," in *12th International Conference on Computer Aided Verification*, Chicago, Illinois, July 2000, pp. 358–372.

[69] C. Baier, J.-P. Katoen, and H. Hermanns, "Approximate symbolic model checking of continuous-time Markov chains," in *10th International Conference on Concurrency Theory (CONCUR'99)*, Eindhoven, The Netherlands, Aug. 1999, pp. 146–162.

[70] B. R. Haverkort, H. Hermanns, and J.-P. Katoen, "On the use of model checking techniques for dependability evaluation," in *19th IEEE Symposium on Reliable Distributed Systems*, Erlangen, Germany, Oct. 2000, pp. 228–237.

[71] E. Clarke, E. Emmerson, and A. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, pp. 244–263, 1986.

[72] M. Ajmone Marsan, G. Balbo, G. Chiola, and G. Conte, "Generalized stochastic Petri nets revisited: Random switches and priorities," in *Proc. 2nd International Workshop on Petri Nets and Performance Models (PNPM '87)*, Madison, Wisconsin, Aug. 1987, pp. 44–53.

[73] G. Chiola, M. Ajmone Marsan, G. Balbo, and G. Conte, "Generalized stochastic Petri nets: A definition at the net level and its implications," *IEEE Transactions on Software Engineering*, vol. 19, pp. 89–107, Feb. 1993.

[74] G. Ciardo and R. Zijal, "Well-defined stochastic Petri nets," in *Proc. 4th International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'96)*, San Jose, California, Feb. 1996, pp. 278–284.

[75] A. Movaghar and J. F. Meyer, "Performability modeling with stochastic activity networks," in *Proc. 1984 Real-Time Systems Symposium*, Austin, Texas, Dec. 1984, pp. 215–224.

[76] M. A. Qureshi, W. H. Sanders, A. P. A. van Moorsel, and R. German, "Algorithms for the generation of state-level representations of stochastic activity networks with general reward structures," in *Sixth International Workshop on Petri Nets and Performance Models*, Durham, North Carolina, Oct. 1995, pp. 180–190.

[77] W. H. Sanders, "Construction and solution of performability models based on stochastic activity networks," Ph.D. dissertation, University of Michigan, Ann Arbor, Michigan, 1988.

[78] G. Ciardo and A. S. Miner, "A data structure for the efficient Kronecker solution of GSPNs," in *Petri Nets and Performance Models (PNPM'99)*, Zaragoza, Spain, Sept. 1999, pp. 22–31.

[79] G. Ciardo and K. S. Trivedi, "A decomposition approach for stochastic Petri net models," in *Proceedings of the Fourth International Workshop on Petri Nets and Performance Models (PNPM '91)*, Melbourne, Victoria, Australia, Dec. 1991, pp. 74–83.

[80] D. Daly and W. H. Sanders, "A connection formalism for the solution of large and stiff models," in *Proceedings of the 34th Annual Simulation Symposium*, Seattle, Washington, Apr. 2001, pp. 258–265.

[81] V. Mainkar and K. S. Trivedi, "Fixed point iteration using stochastic reward nets," in *Proceedings of the Sixth International Workshop on Petri Nets and Performance Models (PNPM '95)*, Durham, North Carolina, Oct. 1995, pp. 21–30.

[82] B. Wichmann, Convener, *Extended Pascal ISO 10206:1990*. ISO/EIC, 1991.

[83] D. Daly, D. D. Deavours, J. M. Doyle, A. J. Stillman, and W. H. Sanders, "Möbius: An extensible framework for performance and dependability modeling," in *Tool Descriptions of the 8th International Workshop on Petri Nets and Performance Models (PNPM '99)*, Zaragoza, Spain, Sept. 1999.

[84] D. Daly, D. D. Deavours, J. M. Doyle, A. J. Stillman, P. G. Webster, and W. H. Sanders, "Möbius: An extensible framework for performance and dependability modeling," in *Digest of Fast Abstracts presented at the 29th Annual International Symposium on Fault-Tolerant Computing (FTCS '29)*, Madison, Wisconsin, June 1999, pp. 15–16.

[85] J. M. Doyle, "Abstract model specification using the Möbius modeling tool," M.S. thesis, University of Illinois at Urbana-Champaign, 1999.

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

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

[88] A. Williamson, "Discrete event simulation in the Möbius modeling framework," M.S. thesis, University of Illinois at Urbana-Champaign, 1998.

[89] J. Sowder, "State-space generation techniques in the Möbius modeling framework," M.S. thesis, University of Illinois at Urbana-Champaign, 1998.

VITA

Daniel Duane Deavours was born in Bloomington, Illinois, in 1972. He received his B.S. in Computer Engineering with honors in January, 1995, and his M.S. in Electrical Engineering in 1997. He received both degrees from the University of Illinois at Urbana-Champaign. He will receive his Ph.D. in Electrical Engineering in 2001, also from the University of Illinois at Urbana-Champaign.

During his graduate studies at the University of Illinois, Dan participated in a number of projects. He maintained the *UltraSAN* modeling tool, and participated in the design of the Möbius modeling tool. He co-designed two short courses for the Motorola University, and co-taught one of the courses.

Dan has authored eleven conference papers and two journal articles. Two of his conference papers have received awards as the best papers at a conference. He joined the faculty at the University of Kansas in August, 2001, as a Research Assistant Professor.