Contents

Preface ix

I Modeling Background xi

1 Möbius Tool 1
   1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
   1.2 Möbius Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
      1.2.1 Framework components . . . . . . . . . . . . . . . . . . . 3
      1.2.2 Tool description . . . . . . . . . . . . . . . . . . . . . . . 4
      1.2.3 State variables . . . . . . . . . . . . . . . . . . . . . . . . 5
      1.2.4 Actions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
      1.2.5 Firing time distributions for actions . . . . . . . . . . . . 9
   1.3 Model Types and Model Solutions . . . . . . . . . . . . . . . . . 20
      1.3.1 Atomic models . . . . . . . . . . . . . . . . . . . . . . . . . 20
      1.3.2 Composed models . . . . . . . . . . . . . . . . . . . . . . 20
      1.3.3 Reward models . . . . . . . . . . . . . . . . . . . . . . . . . 22
      1.3.4 Studies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
      1.3.5 Solution techniques . . . . . . . . . . . . . . . . . . . . . . 22

II Building Models with Möbius 25

2 Project Manager 27
   2.1 Project Menu . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
      2.1.1 Preferences Dialog . . . . . . . . . . . . . . . . . . . . . . . 30
   2.2 Tools Menu . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
   2.3 Help Menu . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
   2.4 Console View . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
CONTENTS

3 Project Editor ............................................. 37
  3.1 Tree View ............................................. 37
  3.2 Project Operations .................................... 38
  3.3 Project Directory Structure .......................... 40
  3.4 Constraints Checker ................................. 40

4 Model Editors ............................................ 43
  4.1 Common Menus ........................................ 43
    4.1.1 File ........................................... 43
    4.1.2 Edit ........................................... 44

5 Atomic Formalisms ....................................... 49
  5.1 SAN .................................................. 49
    5.1.1 SAN primitives ................................ 49
    5.1.2 Editor ........................................... 52
    5.1.3 Edit ............................................. 53
    5.1.4 View ............................................. 56
    5.1.5 Elements ........................................ 57
    5.1.6 Useful techniques ................................ 67
  5.2 Buckets and Balls .................................... 68
    5.2.1 Buckets and Balls primitives .................... 68
    5.2.2 Editor ........................................... 69
    5.2.3 Edit ............................................. 69
    5.2.4 View ............................................. 69
    5.2.5 Elements ........................................ 70
  5.3 PEPA ................................................. 75
    5.3.1 The PEPA \(k\) language ........................ 75
  5.4 Fault Trees ........................................... 77
    5.4.1 Fault tree primitives ............................ 78
    5.4.2 Editor ........................................... 79
    5.4.3 Edit ............................................. 79
    5.4.4 View ............................................. 79
    5.4.5 Elements ........................................ 79
  5.5 External Atomic Interface ............................ 82

6 Composition Formalisms ................................ 83
  6.1 Replicate/Join ........................................ 84
    6.1.1 Overview ....................................... 84
    6.1.2 Replicate/Join composed model editor .......... 84
  6.2 Graph ................................................ 92
6.2.1 Overview .................................................. 92
6.2.2 Graph composed model editor ....................... 93
6.3 Action Synchronization ..................................... 94
  6.3.1 Overview .................................................. 94
  6.3.2 Action Synchronization model editor ............... 94
7 Reward Formalisms ........................................... 97
  7.1 Performance Variables ................................... 97
    7.1.1 Variable definition .................................. 98
    7.1.2 Submodels ............................................. 99
    7.1.3 Rate rewards ......................................... 99
    7.1.4 Impulse rewards ...................................... 100
    7.1.5 Time ................................................... 101
    7.1.6 Simulation ............................................ 103
8 Study Editors .................................................. 105
  8.1 Range Study ............................................... 105
    8.1.1 Incremental range .................................... 106
    8.1.2 Functional range ..................................... 107
    8.1.3 Manual range ......................................... 109
    8.1.4 Random range ........................................ 109
  8.2 Set Study .................................................. 110
  8.3 Experiment Activator ..................................... 112
9 DOE Study Editor Interface ................................ 115
  9.1 User Interface Flow ...................................... 115
  9.2 Defining a DOE Study .................................... 116
  9.3 DOE Study Editor Window ............................... 121
  9.4 Design Tree .............................................. 122
  9.5 Menus .................................................... 124
  9.6 Design Panel ............................................. 126
  9.7 Analysis Procedure ...................................... 128
  9.8 Model Selection .......................................... 132
    9.8.1 Factorial model selection ............................ 132
    9.8.2 ANOVA ............................................... 139
  9.9 Diagnostic Plots ......................................... 143
  9.10 Prediction ............................................... 151
CONTENTS

III Solving Models with Möbius 155

10 How to pick the solver 157

11 Transformers 159
  11.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 159
  11.2 Flat State Space Generator . . . . . . . . . . . . . . . . . . . . 159
    11.2.1 Parameters . . . . . . . . . . . . . . . . . . . . . . . . . 160
    11.2.2 Generation . . . . . . . . . . . . . . . . . . . . . . . . . 163
    11.2.3 Command line options . . . . . . . . . . . . . . . . . . . 164
  11.3 Symbolic State Space Generator . . . . . . . . . . . . . . . . . 165
    11.3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . 166
    11.3.2 Limitations . . . . . . . . . . . . . . . . . . . . . . . . . 168
    11.3.3 Tips on efficient use . . . . . . . . . . . . . . . . . . . . 168
    11.3.4 Parameters . . . . . . . . . . . . . . . . . . . . . . . . . 169
    11.3.5 Generation . . . . . . . . . . . . . . . . . . . . . . . . . 172
    11.3.6 Command line options . . . . . . . . . . . . . . . . . . . 174

12 Simulator 177
  12.1 Simulation Parameters . . . . . . . . . . . . . . . . . . . . . . . 177
    12.1.1 Study and experiment selection . . . . . . . . . . . . . . 177
    12.1.2 Simulation execution parameters . . . . . . . . . . . . . . 179
    12.1.3 Compilation options . . . . . . . . . . . . . . . . . . . . 181
    12.1.4 Check boxes . . . . . . . . . . . . . . . . . . . . . . . . . 182
  12.2 Network Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . 183
  12.3 Running Simulation . . . . . . . . . . . . . . . . . . . . . . . . . 184
    12.3.1 On-line simulation . . . . . . . . . . . . . . . . . . . . . 185
    12.3.2 Off-line simulation . . . . . . . . . . . . . . . . . . . . . 186
  12.4 Simulation Info . . . . . . . . . . . . . . . . . . . . . . . . . . . 186
  12.5 Simulation Results . . . . . . . . . . . . . . . . . . . . . . . . . 188
  12.6 Command Line Options . . . . . . . . . . . . . . . . . . . . . . . 188

13 Numerical Solvers 191
  13.1 Creating a New Numerical Solver . . . . . . . . . . . . . . . . . 191
  13.2 Choosing the Appropriate Numerical Solver . . . . . . . . . . . 192
  13.3 Detailed Descriptions of Supported Numerical Solvers . . . . . 194
    13.3.1 Common options among all analytic solvers . . . . . . . . 194
    13.3.2 Direct steady-state solver . . . . . . . . . . . . . . . . . 196
    13.3.3 Iterative steady-state solver . . . . . . . . . . . . . . . . 198
    13.3.4 Takahashi steady-state solver . . . . . . . . . . . . . . . 201
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C Software Installation</td>
<td>255</td>
</tr>
<tr>
<td>C.1 Example Models</td>
<td>255</td>
</tr>
<tr>
<td>C.2 Database Installation</td>
<td>255</td>
</tr>
<tr>
<td>C.2.1 Windows instructions</td>
<td>255</td>
</tr>
<tr>
<td>C.2.2 Ubuntu Instructions</td>
<td>257</td>
</tr>
<tr>
<td>C.2.3 Möbius Installation</td>
<td>257</td>
</tr>
<tr>
<td>Bibliography</td>
<td>259</td>
</tr>
<tr>
<td>Index</td>
<td>263</td>
</tr>
</tbody>
</table>
Preface

Möbius\textsuperscript{TM} is a software tool for modeling the behavior of complex systems. The first version was released in 2001 as a successor to the popular and successful \textit{UltraSAN} tool. Although Möbius was originally developed for studying the reliability, availability, and performance of computer and network systems, its use has expanded rapidly. It is now used for a broad range of discrete-event systems, from biochemical reactions within genes to the effects of malicious attackers on secure computer systems, in addition to the original applications.

That broad range of use is possible because of the flexibility and power found in Möbius, which come from its support of multiple high-level modeling formalisms and multiple solution techniques. This flexibility allows engineers and scientists to represent their systems in modeling languages appropriate to their problem domains, and then accurately and efficiently solve the systems using the solution techniques best suited to the systems’ size and complexity. Time- and space-efficient distributed discrete-event simulation and numerical solution are both supported.

We hope you find the manual helpful. It is organized into four main parts. Part I describes general modeling concepts and the Möbius framework. Part II describes how models are built in Möbius and contains descriptions of the parameter values found in each dialog and editor. Part III covers the different solution techniques available in Möbius and discusses the advantages of each one. Part IV contains multiple appendices. Appendix [A] is an annotated bibliography of key modeling-related background material. Appendix [B] illustrates the use of Möbius with a detailed step-by-step example. Appendix [??] contains detailed installation instructions. You should refer to those instructions if the Möbius \texttt{setup} program fails to install Möbius correctly.

The Möbius project is one of the major research projects of the Performability Engineering Research Group (PERFORM) in the Coordinated Science Laboratory at the University of Illinois at Urbana-Champaign. Research on Möbius has been supported by Motorola as part of the Motorola Center for High-Availability System Validation, by the National Science Foundation under Grant 9975019, “An Integrated Framework for Performance Engineering and Resource-Aware Compi-
lation and by DARPA grant DABT63-96-C-0069 on “Quality-Based Reliable Computing.”

For an overview of the Möbius project, go to our website (www.mobius.uiuc.edu). Research papers and theses related to Möbius can be found there. Möbius users can also sign up for membership on a mailing list that provides a forum in which users and developers of Möbius can communicate with each other. Information about other research taking place in the PERFORM group is available at www.mobius.uiuc.edu/.

Möbius is copyrighted software; however, it is free if used for educational and research purposes by academic institutions. It is available to nonacademic users under individual and site license agreements. Contact Prof. William H. Sanders at whs@uiuc.edu for details.

The PERFORM group would like to acknowledge Henrik Bohnenkamp, Ric Klaren, Johan Gorter, Holger Hermanns, and Joost-Pieter Katoen from the University of Twente for their work on integrating MODEST with Möbius, and Peter Buchholz and Carsten Tepper of the University of Dortmund and Peter Kemper of TU Dresden for their work integrating the APNN Toolbox and Möbius. We would like to recognize Holger Hermanns for his work on optimal lumping, Peter Kemper of TU Dresden for his work on the state-level AFI, Markus Siegle and Kai Lampka of Universität der Bundeswehr München for their work in developing action-synchronization-based composition, and Boudewijn Haverkort and David Janecek of the University of Twente for their work incorporating FiFiQueues with Möbius.

We appreciate your interest in Möbius. If you have any questions while installing or using the software, please email us at mobius-info@crhc.uiuc.edu.

—The PERFORM group, University of Illinois at Urbana-Champaign, May 2004

1 Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.
Part I

Modeling Background
Chapter 1

Möbius Tool

1.1 Motivation

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

As techniques for solving models advanced, formalisms (or formal languages for expressing models) were also developed. Each formalism has its own merits. Some formalisms afford very efficient solution methods; for example, BCMP [3] queuing networks admit product-form solutions, while superposed generalized stochastic Petri nets (SgSPNs) [15] afford Kronecker-based solution methods, and colored GSPNs (CGSPNs) [6] yield state-space reductions. Other formalisms, such as SPNs [25] and SPAs [18], provide a simple elegance in their modeling primitives, while a number of extensions, such as stochastic activity networks (SANs) [24], were developed for compactly expressing complex behaviors.

Along with formalisms, tools have been developed. A tool is generally built around a single formalism and one or more solution techniques, with simulation sometimes available as a second solution method. [29] lists a number of such tools, such as DyQN-Tool+ [17], which uses dynamic queuing networks as its high-level formalism; GreatSPN [7], which is based on GSPNs [2]; UltraSAN [22], which is based on SANs [24]; SPNP [9], which is based on stochastic reward networks [8];
and TANGRAM-II [5], which is an object- and message-based formalism for evaluating computer and communication systems. While all of these tools are useful within the domains for which they were intended, they are limited in that all parts of a model must be built in the single formalism that is supported by the tool. Thus, it is difficult to model systems that cross different domains and would benefit from multiple modeling techniques.

Möbius takes an integrated multi-formalism, multi-solution approach; the goal was to build a tool in which each model formalism or solver was, to the extent possible, modular, in order to maximize potential interaction. A modular modeling tool is possible because many operations on models, such as composition (described later), state-space generation, and simulation are largely independent of the formalism being used to express the model.

This approach has several advantages. First, it allows for novel combinations of modeling techniques. For example, to the best of our knowledge, the Replicate/Join model composition approach of [30] has been used exclusively with SANs. This exclusivity is artificial, and in the Möbius tool, Replicate/Join can be used with virtually any formalism that can produce a labeled transition system, such as PEPA [11].

The ability to add new components benefits researchers and users alike. Researchers can add a new component to the tool and expect it to be able to interact immediately with other components. Additionally, researchers have access to the work of others, and are able to extend and compare techniques. Users benefit by having access to the most recent developments in conjunction with previously existing techniques. They also benefit from having a modular, “toolbox” approach that allows them to choose the most appropriate tool or tools for the job.

1.2 Möbius Overview

The Möbius tool is an environment for supporting multiple modeling formalisms[1]. For a formalism to be compatible with Möbius, the developer must be able to translate a model built in his/her formalism into an equivalent model that uses Möbius components. Since models are constructed in specific formalisms, the expressive advantages of the particular formalisms are preserved. Because all models are transformed into Möbius components, all models and solution techniques in Möbius with compatible properties are able to interact with each other.

[1]Technically speaking, the given definition is the definition of the Möbius framework, which is described in detail in [12]. However, for the sake of simplicity, we use the terms Möbius tool and Möbius framework interchangeably.
1.2. MöBIUS OVERVIEW

1.2.1 Framework components

To define the framework, it is necessary to identify and abstract the common concepts found in most formalisms. It is also necessary to generalize the process of building and categorizing models. The model’s construction process has been divided into several steps. Each step in the process generates a new type of model. The illustration shown in Figure 1.1 highlights the various model types and other components within the Möbius framework.

![Möbius framework components](image)

Figure 1.1: Möbius framework components.

The first step in the model construction process is to generate a model using some formalism. The most basic model in the framework is called an atomic model, and is made up of state variables and actions. State variables (for example, places in the various stochastic extensions to Petri nets, or queues in queuing networks) hold state information about a model, while actions (such as transitions in SPNs or servers in queuing networks) are the mechanism for changing model state.

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 (i.e., atomic or composed models) to form a larger model. This is sometimes used as a convenient technique to make
the model modular and easier to construct; at other times, the ways that models are composed can lead to efficiencies in the solution process. Examples include the Replicate/Join composition formalism [30], in which symmetries may be detected and state lumping may be performed. Although a composed model is a single model with its own state space, it is not a “flat” model. It is hierarchically built from submodels, which largely preserve their formalism-specific characteristics so that the composed model does not destroy the structural properties of the submodels. Note that the compositional techniques do not depend on the particular formalism of the atomic models that are being composed.

After a composed model is created, the next step is to specify some measures of interest on the model using some reward specification formalism, e.g., that is described in [31]. The Möbius tool captures this pattern by having a separate model type, called reward models, that augments composed models with reward variables.

The next step is typically to apply some solver to compute a solution to the reward model. We call any mechanism that calculates the solution to reward variables a solver. The calculation method could be exact, approximate, or statistical. Consequently, a solver may operate on a model independent of the formalism in which the model was constructed, so long as the model has the properties necessary for the solver.

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

1.2.2 Tool description

The Möbius tool ensures that all formalisms translate model components into framework components through the use of the abstract functional interface (AFI) [16]. The AFI provides the common interface between model formalisms and solvers that allows formalism-to-formalism and formalism-to-solver interactions. It uses abstract classes to implement Möbius framework components. The AFI is built from three main base classes: one for state variables, one for actions, and one that defines overall atomic model behavior. Each of these classes defines an interface used by the Möbius tool when building composed models, specifying reward variables, and solving models.
1.2. MÔBIUS OVERVIEW

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

The main user interface for the Möbius tool presents a series of editors that are classified according to model type. Each formalism or solver supported by Möbius has a corresponding editor in the main interface. These editors are used to construct and specify the model, possibly performing some formalism-specific analysis and property discovery, and to define the parameters for the solution techniques. The tool dynamically loads each formalism-specific editor from a java archive (jar file) at startup. This design allows new formalisms and their editors to be incorporated into the tool without modification or recompilation of the existing code, thus supporting the extensibility of the Möbius tool.

Models can be solved either analytically/numerically or by simulation. From each model, C++ source code is generated and compiled, and the object files are packaged to form a library archive. These libraries are linked together along with the tool’s base libraries to form the executable for the solver. The executable is run to generate the results. The base libraries implement the components of the particular model formalism, the AFI, and the solver algorithms. The organization of Möbius components to support this model construction procedure is shown in Figure 1.2.

We believe that the majority of modeling techniques can be supported within the Möbius tool. By making different modeling processes (such as adding measures, composing, solving, and connecting) modular, we can maximize the amount of interaction allowable between these processes. That approach also makes it possible for the tool to be extensible, in that new atomic modeling formalisms, reward formalisms, compositional formalisms, solvers, and connection formalisms may be added independently. All of these features will be discussed in more detail in the remainder of this manual.

The atomic model represents a generalization of multiple modeling formalisms and is one of the main contributions of Möbius. The key elements of atomic models are state variables and actions, which are the subjects of the next two sections.

1.2.3 State variables

A state variable typically represents some portion of the state of a model, and is a basic component of a model. It can represent something as simple as the number of jobs waiting in a queue, or as complex as the state of an ATM switch.

Different formalisms represent state variables differently. For example, SPNs
Figure 1.2: Möbius tool architecture.
1.2. MÖBIUS OVERVIEW

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) [6] have been extended so that tokens can take on a number of different colors at a place, making the value of a colored place a bag or multi-set. Queuing networks with different customer classes can have more complicated notions of state, such as those found in extended queuing networks [33], in which each job (customer) may have an associated job variable, which is typically implemented as an array of real numbers.

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

1.2.4 Actions

An action is the basic model unit that changes the value of state variables in the Möbius framework, and is therefore the basic model unit that changes model state. An action corresponds to a transition in SPNs [25], GSPNs [2], and other extensions; to an action of an SPA (e.g., [18]); to an activity of a SAN [24]; or to a server of a queuing network (e.g., [3]), for example.

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

Finally, like state variables, the action provides a common interface by which other model components (possibly of different formalisms) and solvers may interact in the Möbius framework. This allows for composition by synchronization, as is found in SPAs, stochastic automata networks (e.g., [27]), and superposed GSPNs (e.g., [15,19]).
Figure 1.3: Execution of timed action in a atomic model.
1.2. MÖBIUS OVERVIEW

Enabling and completion rules for actions

The action chosen to complete in a certain configuration is based upon the action’s time distribution function for all the actions that are currently enabled, along with the fact that instantaneous actions have priority over timed actions. Currently Möbius supports the distribution functions discussed in Section 1.2.5.

Figure 1.3 shows the four possible time lines for the execution of a timed action. The shaded areas represent time during which the action is enabled. Each time line shows the action being enabled initially and an action time being scheduled. After the action time in (a), the action completes, and the new configuration of the model is such that the action is not enabled. After the action time in (b), the action completes, and the new configuration of the model is such that the action is still enabled. Before the action can complete in (c), the enabling conditions become false, and the action is aborted. Finally, before the activity can complete in (d), the activity is reactivated and therefore does not complete until its new activity time has elapsed.

1.2.5 Firing time distributions for actions

This section describes the delay distributions that Möbius currently supports for actions. In the context of the Möbius tool, the random variables in this discussion can be thought of as describing the time until an action fires after becoming enabled, assuming the action is not aborted or reactivated.

For each distribution, you will find a table listing some relevant properties of the distribution, such as its mean and variance. For a given family of distributions (e.g., normal or gamma), there are usually several different ways to define, or parameterize, the density function. Thus, because there is often no universally accepted set of parameters for a given distribution, its parameterization may differ from source to source. For this reason, each table includes a listing of alternative parameters that may be used; if it is not readily clear, a mapping from one parameterization to another is also provided to show how to convert between the two. Finally, each table includes the parameters used by Möbius, and how these parameters map to other common parameterizations.

It is worth mentioning that some confusion over a continuous distribution’s parameters may be eliminated if the parameters are identified with their affect on the distribution rather than just with standard Greek letters. That is, if the parameters are defined correctly, regardless of the symbols they are given, they can be classified, on the basis of their physical or geometric interpretation, as being either scale or shape parameters. A scale parameter determines the unit of measurement of the values in the range of the distribution. A change in the scale parameter com-
presses or expands the density of the corresponding distribution without affecting its general form. A shape parameter, on the other hand, determines the basic form or shape of a distribution within the general family of distributions. A change in the shape parameter fundamentally affects a distribution’s properties (e.g., skewness). Some distributions, such as the exponential and normal, do not have a shape parameter, while others may have several (the beta has two).

More information about these distributions can be found in [1].

**Binomial**

A discrete random variable has a binomial distribution with parameters \((n,p)\), where \(n\) is a positive integer and \(0 < p < 1\), if it represents the number of successes that occur in \(n\) independent trials, each of which results in a success with probability \(p\) and in a failure with probability \(1 - p\). The probability mass function of a binomial random variable is given by

\[
f(k) = \binom{n}{k} p^k (1 - p)^{n-k} \quad k = 0, 1, \ldots, n
\]

As \(k\) goes from 0 to \(n\), \(f(k)\) first increases monotonically and then decreases monotonically, reaching a maximum when \(k = \lfloor (n + 1)p \rfloor\). Also, note that when \(p = \frac{1}{2}\), the distribution is symmetric about its mean (i.e., \(f(k) = f(n - k)\)). When \(p > \frac{1}{2}\), the distribution is skewed to the right; it is skewed to the left when \(p < \frac{1}{2}\).

When \(n\) is large, a binomial random variable with parameters \((n,p)\) can be approximated by a continuous random variable having a normal distribution with the same mean and variance. This is known as the DeMoivre-Laplace limit theorem, and is actually a special case of the Central Limit Theorem. The normal approximation to the binomial will, in general, be quite good for \(n\) large enough that \(np(1 - p) \geq 10\).

<table>
<thead>
<tr>
<th>Mean</th>
<th>Variance</th>
<th>Alternative Parameters</th>
<th>Parameters in Möbius</th>
</tr>
</thead>
<tbody>
<tr>
<td>(np)</td>
<td>(np(1 - p))</td>
<td>((t,p))</td>
<td>((T, P)\rightarrow(n,p))</td>
</tr>
</tbody>
</table>
1.2. MÖBIUS OVERVIEW

Deterministic

An action with a deterministic delay will fire at the time indicated by its parameter, with probability 1. That is, there is no randomness to the firing time. Formally, if this deterministic time is $T > 0$, the density function could be written as

$$f(x) = \delta(x - T)$$

where $\delta$ is the Dirac delta function.

Table 1.2: Summary of deterministic action.

<table>
<thead>
<tr>
<th>Mean</th>
<th>Variance</th>
<th>Alternative Parameters</th>
<th>Parameters in Möbius</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>0</td>
<td>none</td>
<td>Value $\rightarrow T$</td>
</tr>
</tbody>
</table>

Gamma

A random variable is said to have a gamma distribution with shape parameter and rate parameter $(\alpha, \lambda)$, both positive, if its density function is given by

$$f(x) = \begin{cases} \frac{\lambda e^{-\lambda x} (\lambda x)^{\alpha-1}}{\Gamma(\alpha)} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

where $\Gamma$ is the gamma function defined as

$$\Gamma(\alpha) = \int_0^\infty e^{-y} y^{\alpha-1} dy$$

The gamma distribution is often parameterized with a shape parameter and a scale parameter, which is the reciprocal of the rate parameter. Thus in Table 1.3, $\beta = s = b = \frac{1}{\lambda}$.

Exponential

An exponential random variable with parameter $\lambda > 0$ has a probability density function

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases}$$ (1.1)
Table 1.3: Summary of gamma distribution.

<table>
<thead>
<tr>
<th>Mean</th>
<th>Variance</th>
<th>Alternative Parameters</th>
<th>Parameters in Möbius</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{\alpha}{\lambda}$</td>
<td>$\frac{\alpha}{\lambda^2}$</td>
<td>$(\alpha, \beta); (a,s); (k,b); (\gamma, \beta)$</td>
<td>$(\text{Alpha, Beta}) \rightarrow (\alpha, \frac{1}{\lambda})$</td>
</tr>
</tbody>
</table>

The exponential distribution often arises in practice as the distribution of the waiting time until some event occurs, when the time until the event occurs does not depend on how long the wait has been. This is known as the memoryless property. In the context of action firing, the memoryless property states that the probability that an action fires in the next $s$ time units given that it has been enabled for $t$ time units is the same as the initial probability that the action would fire in the first $s$ time units. Mathematically, if $X$ is an exponential random variable representing the time until an action fires after becoming enabled

$$P\{X < s + t | X > t\} = P\{X < s\} \quad \text{for all } s, t \geq 0$$

The exponential distribution is unique in that it is the only continuous distribution possessing the memoryless property. It is this property that permits a Markovian solution of models whose actions are exponential. This will be discussed in more detail later. Note also that the exponential distribution is a special case of the gamma distribution when $\alpha = 1$. As with the gamma distribution, the exponential may be parameterized with $\beta = \frac{1}{\lambda}$ (see Table 1.4). Here $\beta$ is the reciprocal of the rate $\lambda$ and represents the mean of the exponential random variable.

Table 1.4: Summary of exponential distribution.

<table>
<thead>
<tr>
<th>Mean</th>
<th>Variance</th>
<th>Alternative Parameters</th>
<th>Parameters in Möbius</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{1}{\lambda}$</td>
<td>$\frac{1}{\lambda^2}$</td>
<td>$\beta$</td>
<td>Rate $\rightarrow \lambda$</td>
</tr>
</tbody>
</table>

**Erlang**

The Erlang distribution with parameters $(n, \lambda)$ is a special case of the gamma distribution when $\alpha$ is a positive integer ($\alpha = n$). Since $\Gamma(\alpha) = (\alpha - 1)!$ for integral
values of $\alpha$, it follows that the density function of the Erlang distribution is given by

$$f(x) = \begin{cases} \frac{\lambda e^{-\lambda x}(\lambda x)^{n-1}}{(n-1)!} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

Note that when $n = 1$ this distribution degenerates to the exponential distribution. This fact leads to another interpretation of the Erlang distribution. That is, it represents the distribution of the sum of $n$ independent, identically distributed exponential random variables (with parameter $\lambda$). Thus, the Erlang distribution may arise as the waiting time until $n$ events occur, when the time between events is exponentially distributed (e.g., a Poisson process).

<table>
<thead>
<tr>
<th>Mean</th>
<th>Variance</th>
<th>Alternative Parameters</th>
<th>Parameters in Möbius</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{n}{\lambda}$</td>
<td>$\frac{n}{\lambda^2}$</td>
<td>$(k,\lambda);(m,\beta)$</td>
<td>$(M,Beta)\rightarrow(n,\frac{n}{\lambda})$</td>
</tr>
</tbody>
</table>

**Beta**

A random variable has a beta distribution with parameters $\alpha$ and $\beta$, both positive, if its probability density function is given by

$$f(x) = \begin{cases} \frac{1}{B(\alpha, \beta)} x^{\alpha-1} (1 - x)^{\beta-1} & 0 < x < 1 \\ 0 & \text{otherwise} \end{cases}$$

where $B$ is the beta function given by

$$B(\alpha, \beta) = \int_0^1 x^{\alpha-1} (1 - x)^{\beta-1} dx$$

The beta function can also be written in terms of the gamma function, which was defined previously.

$$B(\alpha, \beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha + \beta)}$$

The beta distribution can be used when the firing time of an action can take on values in some finite interval $[c,d]$, which can be mapped to the interval $[0,1]$ by letting $c$ denote the origin and taking $d - c$ as a unit length. When $\alpha = \beta$, the beta
density is symmetric about $\frac{1}{2}$, putting more probability mass in the region about $\frac{1}{2}$ as the common value for the parameters increases. When $\alpha > \beta$, the density is skewed to the right (meaning that larger values are more likely), and it is skewed to the left when $\alpha < \beta$.

Table 1.6: Summary of beta distribution.

<table>
<thead>
<tr>
<th>Mean</th>
<th>Variance</th>
<th>Alternative Parameters</th>
<th>Parameters in Möbius</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>$\frac{\alpha \beta}{(\alpha + \beta)(\alpha + \beta + 1)}$</td>
<td>$(a,b)$</td>
<td>(Alpha1, Beta1) $\rightarrow$ $(\alpha, \beta)$</td>
</tr>
</tbody>
</table>

Hyperexponential

Let $X_i, i = 1, \ldots, n$, be $n$ independent exponential random variables, each with parameter $\lambda_i$, where $\lambda_i \neq \lambda_j$ for $i \neq j$. Suppose also that there are $n$ positive constants $p_i$ such that $0 < p_i < 1$ for $i = 1, \ldots, n$ and $\sum_{i=1}^{n} p_i = 1$. If the random variable $X = X_i$ with probability $p_i$, then $X$ is a hyperexponential random variable with $n$ exponential stages and parameters $(p_i, \lambda_i), i = 1, \ldots, n$. That is, a hyperexponential random variable is a probabilistic choice among exponentials with different rates. In Möbius, the hyperexponential distribution for actions has $n = 2$.

Formally, a hyperexponential random variable $X$ has a probability density function defined as

$$f_X(x) = \sum_{i=1}^{n} p_i f_{X_i}(x)$$

where $f_{X_i}$ is the probability density function for an exponential random variable with parameter $\lambda_i$, given by Equation 1.1.

Finally, a note is in order about the variance of the hyperexponential distribution. In general, it is not a weighted sum of the variances, as it is for the mean (see Table 1.7). It can be calculated using the definition of variance as

$$Var(X) = \sum_{i=1}^{n} p_i E[(X_i - E[X])^2]$$

where $E[X] = \sum_{i=1}^{n} \frac{p_i}{X_i}$ as in Table 1.7. It is known, however, that the coefficient of variation (CV), $C_X = \frac{\sqrt{Var(X)}}{E[X]} > 1$. Thus, one of the characteristics of the
**1.2. MÖBIUS OVERVIEW**

Table 1.7: Summary of hyperexponential distribution.

<table>
<thead>
<tr>
<th>Mean</th>
<th>Variance</th>
<th>Alternative Parameters</th>
<th>Parameters in Möbius</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sum_{i=1}^{n} \frac{\bar{X}_i}{\lambda}$</td>
<td>See text</td>
<td>$(p_i, \mu_i); (\alpha_i, \lambda_i)$</td>
<td>(Rate1, Rate2, P) $\rightarrow (\lambda_1, \lambda_2, p_1)^{\dagger}$</td>
</tr>
</tbody>
</table>

Hyperexponential distribution is that it has higher variability than the exponential distribution, which has CV equal to 1.

**Negative Binomial**

A discrete random variable is said to have a negative binomial distribution with parameters $(s, p)$, where $s$ is a positive integer and $0 < p < 1$, if it represents the number of independent trials, each of which has a probability $p$ of success, that must be performed until a total of $s$ successful trials have occurred. The probability mass function of a negative binomial random variable is

$$f(k) = \binom{k-1}{s-1} p^s (1-p)^{k-s} \quad k = s, s+1, \ldots$$

Table 1.8: Summary of negative binomial distribution.

<table>
<thead>
<tr>
<th>Mean</th>
<th>Variance</th>
<th>Alternative Parameters</th>
<th>Parameters in Möbius</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s/p$</td>
<td>$s(1-p)/p^2$</td>
<td>$(r, p)$</td>
<td>$(S, P) \rightarrow (s, p)$</td>
</tr>
</tbody>
</table>

**Geometric**

A random variable has a geometric distribution with parameter $p$, $0 < p < 1$, if it represents the number of independent trials, each of which has a probability $p$

---

$\dagger$Only one $p$ parameter is needed, since $p_2 = 1 - p_1$
of being a success, that must be performed until a success occurs. A geometric random variable has a probability mass function given by

\[ f(k) = (1 - p)^{k-1} p \quad k = 1, 2, \ldots \]

Note that a geometric random variable is just a negative binomial with \( s = 1 \).

The geometric distribution is the only discrete distribution with the memoryless property, and thus a geometric random variable can be thought of as a “discretized” exponential random variable.

Table 1.9: Summary of geometric distribution.

<table>
<thead>
<tr>
<th>Mean</th>
<th>Variance</th>
<th>Alternative Parameters</th>
<th>Parameters in Möbius</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{1}{p} )</td>
<td>( \frac{1-p}{p^2} )</td>
<td>none</td>
<td>P ( \rightarrow p )</td>
</tr>
</tbody>
</table>

Uniform

A random variable is uniform with parameters \( \alpha \) and \( \beta \), \( 0 \leq \alpha < \beta \), if its density function is constant over the interval \((\alpha, \beta)\).

\[ f(x) = \begin{cases} 
\frac{1}{\beta - \alpha} & \alpha < x < \beta \\
0 & \text{otherwise}
\end{cases} \]

The uniform distribution can be used, for example, to model the lifetime of an item that is equally likely to fail at all points in some interval of time. More precisely, the probability that the item will fail in some subinterval \((c,d)\) of \((\alpha, \beta)\) depends only on the length of the subinterval, \(d - c\). Thus, the probability mass is distributed “uniformly” over the interval \((\alpha, \beta)\).

Table 1.10: Summary of uniform distribution.

<table>
<thead>
<tr>
<th>Mean</th>
<th>Variance</th>
<th>Alternative Parameters</th>
<th>Parameters in Möbius</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{\alpha + \beta}{2} )</td>
<td>( \frac{(\beta - \alpha)^2}{12} )</td>
<td>((a,b))</td>
<td>((\text{LBound, UBound}) \rightarrow (\alpha, \beta))</td>
</tr>
</tbody>
</table>
1.2. MÖBIUS OVERVIEW

Triangular

A triangular random variable with parameters \((a, b, c)\), \(0 \leq a < c < b\), has a triangle-shaped density function given by

\[
f(x) = \begin{cases} 
\frac{2}{b-a} \frac{(x-a)}{c-a} & a \leq x \leq c \\
\frac{2}{b-a} \left(1 - \frac{x-c}{b-c}\right) & c < x \leq b \\
0 & \text{otherwise}
\end{cases}
\]

If \(c = \frac{a+b}{2}\), the distribution is symmetric about \(c\) (i.e., there is equal probability mass to the left and right of \(c\)). However, if \(c < \frac{a+b}{2}\) or \(c > \frac{a+b}{2}\), the distribution is skewed to the left or right, respectively.

The triangular distribution could arise, for example, as the sum of two independent uniform random variables. If \(X\) and \(Y\) are independent random variables, both uniformly distributed on \((0,1)\), then \(X + Y\) has a triangular distribution with \(a = 0\), \(b = 2\), and \(c = 1\).

Table 1.11: Summary of triangular distribution.

<table>
<thead>
<tr>
<th>Mean</th>
<th>Variance</th>
<th>Alternative Parameters</th>
<th>Parameters in Möbius</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\frac{a+b+c}{3})</td>
<td>(\frac{a^2+b^2+c^2-ab-bc-ac}{18})</td>
<td>none</td>
<td>((A, B, C) \rightarrow (a, b, c))</td>
</tr>
</tbody>
</table>

Weibull

A random variable whose density function is given by

\[
f(x) = \begin{cases} 
\frac{\alpha}{\beta} \left(\frac{x}{\beta}\right)^{\alpha-1} \exp\left\{-\left(\frac{x}{\beta}\right)^\alpha\right\} & x \geq 0 \\
0 & x < 0
\end{cases}
\]

is said to be a Weibull random variable with shape parameter and scale parameter \((\alpha, \beta)\) where \(\alpha, \beta > 0\). When \(\alpha = 1\) this distribution reduces to the exponential distribution with parameter \(\lambda = \frac{1}{\beta}\).

The Weibull distribution was originally developed for the interpretation of fatigue data, but now it is widely used in engineering practice. In particular, it can arise as the distribution of the lifetime of an object, especially when the “weakest link” model is appropriate for the object (meaning that the object fails when any of its parts fail).
Table 1.12: Summary of Weibull distribution.

<table>
<thead>
<tr>
<th>Mean</th>
<th>Variance</th>
<th>Alternative Parameters</th>
<th>Parameters in Möbius</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \beta \Gamma(1 + \frac{1}{\alpha}) )</td>
<td>( \beta^2 \left[ \Gamma(1 + \frac{2}{\alpha}) - \frac{\beta^2}{\Gamma^2(1 + \frac{1}{\alpha})} \right] )</td>
<td>((\beta, \alpha);(k,b))</td>
<td>((\text{Alpha, Beta}) \rightarrow (\alpha, \beta))</td>
</tr>
</tbody>
</table>

**Conditional Weibull**

A random variable whose density function is given by

\[
f(x) = \begin{cases} \frac{\alpha (\frac{x+t}{\beta})^{\alpha-1} \exp\left\{-(\frac{x+t}{\beta})^\alpha\right\}}{\Gamma(1 + \frac{\alpha}{\beta})}, & x \geq 0, t \geq 0 \\ 0, & x < 0, t \geq 0 \end{cases}
\]

is said to be a Conditional Weibull random variable with shape parameter, scale parameter and elapsed time \((\alpha, \beta, t)\) where \(\alpha, \beta > 0\) and \(t \geq 0\). It is said to be conditional because of the fact that the distribution has already accumulated \(t\) hours of operation successfully. When \(t = 0\), this distribution reduces to Weibull distribution. When \(\alpha = 1\) and \(t = 0\), this distribution reduces to the exponential distribution with parameter \(\lambda = \frac{1}{\beta}\).

Table 1.13: Summary of Conditional Weibull distribution. Note \(\alpha \neq 1\)

<table>
<thead>
<tr>
<th>Mean</th>
<th>Variance</th>
<th>Alternative Parameters</th>
<th>Parameters in Möbius</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{1}{e^{-\left(\frac{x}{t}\right)\beta}} \left[ \frac{\beta^2}{\alpha} \Gamma\left(1 + \frac{2}{\alpha} \right) \left(\frac{2t}{\beta}\right)^\alpha \right] + te^{-\left(\frac{2t}{\beta}\right)\alpha} )</td>
<td>-</td>
<td>((\beta, \alpha, t);(k,b,t))</td>
<td>((\text{Alpha, Beta, t}) \rightarrow (\alpha, \beta, t))</td>
</tr>
</tbody>
</table>

**Normal**

A random variable is normally distributed, with parameters \((\mu, \sigma^2)\), if its density is given by

\[
f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/2\sigma^2}, \quad -\infty < x < \infty
\]
This density function is a bell-shaped curve that is symmetric about $\mu$. The scale parameter $\sigma > 0$ represents the spread of the distribution; smaller values of $\sigma$ correspond to more probability mass around $\mu$.

An important fact about normal random variables is that if $X$ is normally distributed with parameters $(\mu, \sigma^2)$, then $Y = aX + b$ is also normally distributed with parameters $(a\mu + b, a^2 \sigma^2)$. A special case of this observation is when $a = \frac{1}{\sigma}$ and $b = -\frac{\mu}{\sigma}$. Then $Z = \frac{X - \mu}{\sigma}$ is normally distributed with parameters $(0,1)$. Such a random variable $Z$ is said to have the standard normal distribution.

Many random phenomena obey, at least approximately, the normal distribution; thus it arises quite often in practice. Some examples of this behavior include errors of various types and quantities that are the sum of a large number of other quantities (by virtue of the central limit theorem).

One final note is in order about the range of the normal distribution. In general, a normal random variable can take on any real-numbered value, as in Equation 1.2. However, if the normal distribution is used to represent a nonnegative quantity (i.e., time), as in Möbius, then its density should be truncated at $x = 0$ so that only positive quantities are generated. This is done by letting $q = \int_0^\infty f(x)\,dx$ where $f$ is the original density function given in Equation 1.2. Note that $q$ will be less than 1. Then the truncated density function, with range $[0,\infty)$, is given by

$$f^*(x) = \begin{cases} \frac{f(x)}{q} & x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

This truncated density function for $X$ is just a conditional density, conditioned on the event that $X$ is nonnegative. The bell-shaped curve of the original density function is preserved, but its mean and variance are changed.

Table 1.14: Summary of normal distribution.

<table>
<thead>
<tr>
<th>Mean</th>
<th>Variance</th>
<th>Alternative Parameters</th>
<th>Parameters in Möbius</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu$</td>
<td>$\sigma^2$</td>
<td>none</td>
<td>(Mean, Variance)→($\mu, \sigma^2$)</td>
</tr>
</tbody>
</table>

**Lognormal**

A random variable $X$ is lognormally distributed if $Y = \ln(X)$ is normally distributed. The general formula for the density function of the lognormal distribution
The lognormal distribution is commonly used to model the lifetime of objects whose failure modes are of a fatigue-stress nature. Consequently, the lognormal is a good companion to the Weibull distribution when modeling such objects.

Table 1.15: Summary of lognormal distribution.

<table>
<thead>
<tr>
<th>Mean</th>
<th>Variance</th>
<th>Alternative Parameters</th>
<th>Parameters in Möbius</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e^{\mu+\alpha^2/2}$</td>
<td>$e^{2\mu+\alpha^2}(e^{\alpha^2} - 1)$</td>
<td>($m,\sigma$);($\mu,\sigma$)</td>
<td>(Mu,Alpha,Squared)$\rightarrow$(\mu,\alpha^2)</td>
</tr>
</tbody>
</table>

1.3 Model Types and Model Solutions

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

1.3.1 Atomic models

Atomic models are the basic building blocks of a model. Since multiple formalisms can be supported in Möbius, the user can use different formalisms to model different aspects of his/her model. New formalisms are constantly added into the tool as needed.

1.3.2 Composed models

The Möbius framework allows the construction of composed models from previously defined models. That allows the modeler to adopt a hierarchical approach to modeling, by constructing submodels as meaningful units and then placing them
together in a well-defined manner to construct a model of a system. This is accomplished by the state-sharing approach, which links submodels together by identifying sets of state variables. For example, it is possible to compose two Petri net models by holding a particular place in common. That allows for interaction between the submodels, since both can read from and write to the identified state variable. This form of state sharing is known as equivalence sharing, since both submodels have the same relationship to the shared state variable. Note that a composition formalism maintains the dependencies among the actions and the state variables in the composed model based on the sharing approach and dependencies in the underlying submodels. Currently, the Möbius tool features two composed model formalisms that use equivalence sharing: Replicate/Join \[10, 30\] and Graph composition \[10, 26, 36\].

Replicate/Join

The Replicate/Join composed model formalism was originally conceived for SAN models (see \[50\]). It enables the modeler to define a composed model in the form of a tree, in which each leaf node is a predefined atomic or composed model, and each non-leaf node is classified as either a Join node or a Replicate node.

A Join is used to compose two or more submodels using equivalence sharing. A Replicate is used to construct a model consisting of a number of identical copies of its single child. Each child node of a Replicate or Join node can be a Replicate, a Join, or a single atomic or composed model.

Since the instances of a Replicate composed model are indistinguishable, its state can be represented in a lumped way as a sequence of numbers, each denoting the number of instances in each component state. Since the composed model presents its state to the Möbius tool through the AFI, it can keep details of symmetry-based reductions private. The rest of the Möbius tool does not know, and has no need to know, the details of such optimizations.

Graph composition

The Möbius tool supports a second composed model formalism called Graph composition \[26, 36\]. Whereas the structure of a Replicate/Join composed model is a tree, the structure of a Graph composed model is a graph, in which an arc linking two models indicates an equivalence-sharing relationship between the two models. As with Replicate/Join composition, lumping techniques based on computational group theory can be used to find all symmetries in the graph structure of the model automatically \[26\].
1.3.3 Reward models

Reward models [31] build upon atomic and composed models, equipping them with the specification of a performance measure. At this time we have implemented one type of reward model in the Möbius tool: a performance variable (PV). A PV allows for the specification of a measure on one or both of the following:

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

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

1.3.4 Studies

During the specification of atomic, composed, and reward models in the tool, global variables can be used to parameterize model characteristics. A global variable is a variable that is used in one or more models, but not given a specific value. Models are solved after each global variable is assigned a specific value. One such assignment forms an experiment. Experiments can be grouped together to form a study.

1.3.5 Solution techniques

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

---

2Note that although these variables are called performance variables, they are generic and can be used to represent either dependability or performability measures.
are performed by the formalism editor. M"obius allows formalism-specific solution techniques in the form of property-specific solvers. In order to evaluate the overhead resulting from the generality of the M"obius framework, we have compared the performance of the simulators in UltraSAN and M"obius [39]. We observed that the amount of overhead is negligible compared to runtime differences caused by other factors, such as the algorithm used by the solver and the optimization techniques used in the implementation.

**State-space generator**

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

**Simulation**

The M"obius tool currently supports two modes of discrete event simulation: transient and steady-state. In the transient mode, the simulator uses the independent replication technique to obtain statistical information about the specified reward variables. In the steady-state mode, the simulator uses batch means with deletion of an initial transient to solve for steady-state, instant-of-time variables. Estimates available during simulation include mean, variance, interval, and distributions. Confidence intervals are computed for all estimates.

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

Building Models with Möbius
Chapter 2

Project Manager

The Project Manager is the main console for Möbius and is shown in Figure 2.1. Across the top is a main menu, with three menu categories: PROJECT, TOOLS, and HELP. The main display is a console view that shows messages for the operations performed by Möbius. Descriptions of the operations available under each menu category are presented in the following sections.
2.1 Project Menu

The PROJECT menu contains several operations related to Möbius projects. A Möbius project is the fundamental unit of organization for each system model in Möbius. A project contains one or more models defining the system and instances, descriptions of what parameters to solve for, and instances of solvers to generate the solutions. More details on projects can be found in Chapter 3.

The PROJECT menu contains the following operations:

- **NEW**: Create a new project. A dialog will appear prompting for the new project name. After you enter a valid project name, the author, and a description. The author and description are optional and for informational purposes only. These two fields can be changed at a later date. The new project will be opened.

- **OPEN**: Open an existing project. A dialog to select the project to open will appear as shown in Figure 2.2. You may select multiple projects to be opened simultaneously. You can quickly open a single project by double clicking it.

![Open Projects](image)

Figure 2.2: Use the project selection dialog to open projects.

- **COPY**: Copy an existing project to a new project. You must close the existing project to guarantee a stable copy.
2.1. PROJECT MENU

- **RENAME**: Give an existing project a new name. You must close the project before it can be renamed.

- **CLEAN**: Remove all temporary files from the project.

- **RESAVE**: Regenerate all project source code and libraries. Resaving the project is required before a project is opened for the first time after a Möbius upgrade. You must also resave after unarchiving a project.

- **DELETE**: Delete the project. The most-recently-deleted project is stored in a “Deleted” folder within the Möbius project directory.

- **ARCHIVE**: Create a backup of the project. After you select one or more projects, you will be asked for details about how you want each of the selected projects to be archived. After you specify the name of the archive file, you must select one of two options, **Full archive** and **Minimal archive**. A **full archive** includes all files found within the project. A **minimal archive** includes only the files necessary to define all models within the project. In a minimal archive, object files, libraries, executables, and results are not backed up. Minimal archives are helpful when multiple users are collaborating on the same model, since they create small archives that are convenient to email. The format for the archive is gzipped tar (.tgz) format.

- **UNARCHIVE**: Restore a project from an archive file. If the project to be restored has the same name as an existing project, you can either delete the existing project or cancel the restoration. Restoring one project on top of a different one could cause corruption and confusion, so is not supported. Unarchived projects should be resaved before they are used.

- **DOCUMENT**: Create HTML documentation for the chosen project. After you select the project to document, you use the **Document Project** dialog (see Figure 2.3) to select the models in the project that should be documented. Documentation is written to a subdirectory in the **Documentation** directory within the selected project’s project directory. The name of the subdirectory is given by the **Document Name** field in the dialog. You can view it with a web browser, or import it into software such as Microsoft Word for format refinement and printing.

Möbius supports three image formats for output: PNG, SVG, and PDF. You select which formats you want generated by selecting the appropriate checkboxes. Also, you can include one of these outputs into the generated HTML documentation.
There are four options in the dialog that are helpful when dealing with images that are too large to fit on a single page:

- **Generate one complete image** will create a single image for the model.
- **Generate subimages if necessary** will subdivide the full image in width and/or height and generate two or four subimages. The width is subdivided if the original image is more than 650 pixels wide. The height is subdivided if the original image is more than 700 pixels tall.
- **Scale subimages** will scale each subimage that is larger than a single page in size. The scaling is proportional and the resulting image will be at most 650 wide or 700 pixels tall.

- **PREFERENCES**: Launch the preferences window (see Figure 2.4).
- **QUIT**: Exit Möbius.

### 2.1.1 Preferences Dialog

The **PREFERENCES** dialog (Figure 2.4) provides access to various preferences and settings used during the operation of the Möbius tool. The preferences are broken down into pages listed on the left side of the dialog. Above the pages is a filter text box that you can quickly type a keyword in and the preference pages pertinent to that keyword will be displayed. For example, typing “user” in the filter box will eliminate all the pages except the DATABASE page, because it contains a username field used by the database connection feature.

The **PREFERENCES** dialog contains the following page:

- **BUILD**: This preference page allows you to define the default compile mode and default simulation execution architecture. Möbius can compile in two modes: *normal* and *optimized*. Normal mode incorporates compiler-created debugging symbols into the object files, and uses debug-enabled Möbius AFI libraries when creating the executables, allowing project-generated executables (such as the simulator and state-space generator) to be run through C++ debuggers like gdb. Normal mode also enables Möbius trace output, making it possible to generate files that trace the progress of the model during solution. Normal mode object code takes less time to generate, but is often much larger than optimized code, due to the extra debugging symbols that are included.
Figure 2.3: The Document Project dialog is used to select models from the project to include in the documentation.
Figure 2.4: The Möbius Preferences dialog is used to change various preferences about the operation of Möbius.

Optimized mode enables full compiler optimizations, and disables debug symbols and trace output. Optimized mode should be used when the model is running well, and results need to be produced rapidly.

The results generated should be the same in either mode. The modes allow for the trade-off between extra information needed for debugging and speed of data production.

The default simulation execution architecture mode has two options: 32-bit and 64-bit. For machines running a 32-bit version of their operating system, this setting has no effect as simulations will always be executed in 32-bit mode. However, for machines running a 64-bit operating system, you have the option of building and executing 32-bit or 64-bit simulations. 32-bit simulations will likely run a little faster and take up significantly less memory during execution. However, 64-bit simulations enable access to greater amounts of memory space that may be necessary for large simulations. Simulation results in either mode should be the same.

- DATABASE: The DATABASE settings dialog displays the list settings that are needed to interface Möbius the integrated results database features with the external database server.
2.1. PROJECT MENU

Figure 2.5: The Results Database preference page is used to specify the database access settings.

- **Database Enabled** determines if Möbius will connect to the database server using the specified configuration settings.
- **User Name** specifies the name of the database user that Möbius should use when connecting to the database system.
- **Password** specifies the password for the indicated database user.
- **Database** is the name of the database Möbius will create. Multiple databases can be created as needed, simply by changing this name.
- **Hostname** is the name of the machine where the database server is located. Can be a full machine name or 'localhost'.
- **Port** is the listening port of the database server. The default port is 5432. Ask your system administrator if the port was changed when the database server was configured.

**EXTERNAL TOOLS:** The External Tools preference page lists the external tools Möbius uses during its operation. In special cases, you may want to change the tool used to perform a certain operation (for example, the remote shell or archive command). The provided tool must either be in your PATH or the absolute path to the tool should be provided. Some fields are used to specify additional arguments to the external tool whenever it is called.
– **Compiler**: The G++ compiler that Möbius should use.

– **Compiler Options**: Extra compiler options to be used when compiling the Möbius project. This field was originally used to add debugging flags. In current versions, debugging flags are automatically added as part of the normal compilation mode, and this field should typically be blank.

– **Linker**: The linker, ld. Usually set to the path to the standard linker for the system.

– **Linker Options**: Extra linker options to be used when linking Möbius solvers (such as the simulator and state-space generator). For standard use, this field can be left blank.

– **Make Command**: The GNU make tool. The version of make should be 3.7x or later.

– **Archive Command**: The ar tool, the system command used to create C++ library files.

– **Ranlib Command**: The ranlib tool. Ranlib is used during the creation of project libraries.

– **Remote Shell Command**: The command used to gain remote access to other machines on the network. It is not necessary to specify a remote shell command to run Möbius on a single local machine. Typically this command is either rsh or ssh. In either case, system access privileges must be set so that the command can execute without entering a password or pass phrase. More details are available in Appendix section ?? and in the system man pages for rsh and ssh.

– **Tar Command**: The GNU tar command. This command is used to create and restore archive files. NOTE: In the current version of Möbius, tar is used with the “z” option, specifying the archive to be compressed with gzip. Typically gzip is installed in the same directory as tar, but most implementations of tar expect to find gzip in the PATH, and do not expect to have to look for it. If there are error messages when “archive” or “unarchive” commands are run, please see if gzip is found in the system PATH, and update the PATH if necessary.

• **LOCATIONS**: The locations preference page contains a list of system paths necessary for the operation of Möbius.

– **Möbius Installation Path**: The directory where the Möbius software is installed. This field should update automatically, based on the location
2.2 Tools Menu

The TOOLS menu lists the various utilities available in Möbius. The DATA MANAGER menu item will launch the Möbius Data Manager, which is further described in Chapter [15].

2.3 Help Menu

The HELP menu provides various help for the use of Möbius. The ABOUT menu item opens the About window. The About window contains information about this version of Möbius, including the vendor and copyright statement.

2.4 Console View

The Console View provides a running output of the operations of Möbius. When a project is opened or a model component is saved, this event is logged by the console for your review. The Console View has several icons in the upper right corner to control the behavior of the console and to switch between the available console logs.

The first icon looks like a document with a grey X over it. This icon clears all the text from the current console log. This can be useful for when you want a fresh start to the log.

The next icon has a lock over a scrolling control. This is the scroll lock and will prevent the console from automatically scrolling down as new text is appended to the console.

The next icon has pin on a window icon. This is the pin console button and will keep the console view from switching to a different console automatically.
Finally, the last icon allows you to switch between the available console logs. The Main console provides general information about the operation of Möbius. The Error console provides detailed error messages when something goes wrong. The Detailed console provides the finest grain of detail messages possible and includes messages from the other two consoles. Of particular interest, when Möbius executes an external command (e.g. make, java, etc.) the command and its arguments are logged in the Detailed console.

The text of a console log can be highlighted for copy/paste operations like you would expect. Also, if you right click on the text, a simple find feature is available to quickly search through the contents of the log.
Chapter 3

Project Editor

The Möbius project is the fundamental unit of organization for each system model in Möbius. A complete project will contain the following: one or more models defining the system behavior, a description of what to measure about the system, specification of all model parameters, at least one instantiation of a solver to compute the required measurements, and the results produced by the solver.

3.1 Tree View

The model types in the project are classified in the same hierarchy as the components of the Möbius framework (see Chapter 1). The model types that may exist in a project are listed below.

- Atomic models describe the behavior of individual components of the system.
- Composed models are hierarchical models used to combine atomic and composed models to form the complete model of the system.
- Reward models define measures of interest about the system.
- Study models create experiments that specify values for the system parameters in the model.
- Solvers solve the model using simulation, or prepare the model for numerical solution using state-space generation.
- Numerical solvers solve the model using a variety of numerical algorithms.
CHAPTER 3. PROJECT EDITOR

Figure 3.1: Tree view in the project editor, with Atomic selected.

The model types are arranged in a tree, in which the names of the various classifications are the main nodes, and the instances of each classification are the children of the node. The tree view from an example project editor is shown in Figure 3.1.

The order of model types in the tree represents the typical dependency among models in a project. A model of a given type typically depends on models of the type one level above it in the tree. For example, a study model depends on a reward model. When one model depends on another, it is often referred to as the parent, and the model it depends on is referred to as the child.

3.2 Project Operations

Several operations are possible in the project editor. You can access the operations via the toolbar or through a pop-up menu activated by right-clicking within the tree view.

NEW To create a new model within a project, select the desired category and click on NEW. A dialog will appear, asking for the type of formalism to create. For example, Figure 3.2(a) displays the possible types of atomic models that can be created. The dialog also gives the opportunity to name the new model.

After you click the OK button, the new model editor will open. In the case of the reward, study, solver, and numerical solver models, another pop-up dialog will immediately appear, requesting the name of the child model of the new model, as
3.2. PROJECT OPERATIONS

(a) Specify the name and type of newly created models.
(b) Specify the newly created model’s child model.

Figure 3.2: Dialogs shown when a new Möbius model is created.

shown in Figure 3.2(b).

OPEN Select a model name from the tree view, and then click OPEN to open it. Multiple models may be open at the same time.

COPY Select a model name or type from the tree view, and then click COPY to generate a copy of the model. Another dialog opens, allowing you to copy the selected model to another file within this project, or copy it into another project. Also, you can copy models from another project into the current one.

IMPORT Select a model type in the tree view and then click IMPORT to import this type of model from an existing UltraSAN project. A dialog will appear, listing all the UltraSAN projects found in the UltraSAN project directory (see Section ?? on how to set the UltraSAN project directory). After you select one of the UltraSAN projects, another dialog will appear listing all possible models in the UltraSAN project of the type selected in Möbius.

\[^1\text{UltraSAN was the predecessor to Möbius.}\]
CHAPTER 3. PROJECT EDITOR

Figure 3.3: An instance of the copy dialog, which is used to copy models within a project, or from one project to another.

**DELETE**  You can delete a model in a Möbius project by selecting the model in the tree view and choosing the DELETE menu.

**CLOSE**  You can close the project editor using the Close button, after all of the model editors of the project have been closed.

### 3.3 Project Directory Structure

Each project is stored in a directory named after the project within the Möbius project directory. It is necessary to understand the basic structure of a project directory, since Möbius stores output files, such as solution results or model documentation, in various locations within the project subtree. Files within the project directory are arranged in the same tree structure as the tree view.

Solution results and other solver outputs are typically written to the model directory corresponding to the solver that generated the results.

All documentation is stored in a subdirectory of the Documentation directory in the top-level directory of the project. The name of this subdirectory is given by the Document Name field in the Document Project dialog. See Fig 2.3 for an example.

The project and each model in the project are defined in separate files, which are also named after the project or individual model and have extensions unique to each type of model. The files are XML-based, and under most circumstances should not be edited directly. Doing so can corrupt the Möbius project.

### 3.4 Constraints Checker

When a project, or a submodel within the project is saved in the Project Editor, the **constraints checker** is called by the Project Manager to ensure that the model,
as saved, doesn’t violate any conditions imposed upon it by solver modules which utilize the model. If the constraints checker detects a violation of a constraint, it produces a pop-up warning informing the user of the problem, prompting them to correct the issue.

For more on specific constraints for individual solvers, see Chapter [3]
Chapter 4

Model Editors

4.1 Common Menus

There are several menus that are common to each model editor in Möbius. Instead of describing these features repeatedly, the functionality common to each model editor is discussed in this chapter.

4.1.1 File

The FILE menu is common to all editors, and typically contains the following common commands:

- **SAVE**: Store the current model to disk, and then translate the model into the appropriate C++ code and compile the code to form the representative library for this model.

- **SAVE NO COMPILE**: Store the current model to disk, but avoid performing the compilation. It is still necessary to compile the model, either by selecting SAVE from the FILE menu, or by compiling within the solver.

- **CLOSE**: Close the editor. If there are unsaved changes, a dialog will appear, prompting you to save before closing.

- **DOCUMENT MODEL**: Create HTML documentation for the current model. The documentation files will be stored in a subdirectory in the Document directory within the project’s project directory. The name of the subdirectory will be what you specify in the Document Name field of the dialog box. See Fig. 2.3.
**4.1.2 Edit**

**Undo**

You are allowed to undo most recent operations on the model using the **UNDO** menu. Examples of operations that can be revoked using the undo feature include the movement of graphical components in an editor, the renaming of components, the editing of text, the addition of new components, and the addition of lines. The **UNDO** menu will list the name of the most recently completed operation that can be undone. Undo operations are stored so that multiple operations can be undone.

**Edit global variables**

![Global Variables editor](image)

Figure 4.1: **Global Variables** editor.

The behavior of complex systems being modeled often depends on several parameters. Often these parameters are unknown, or can take on multiple values, depending on the configuration of the system. Möbius provides the option of using global variables instead of pure numerical values to define the different characteristics of elements like state variables and actions. The values of the global variables are then set by experiments in the study model.

In each atomic or composed model, you can add or delete global variables by clicking on the **Edit Global Variables** option on the menu editor. A pop-up dialog box, as shown in Figure 4.1, is used to add, delete, and view global variables. The dialog box can remain open as a reference while you are editing the model.

- The **Add** button is used to create new global variables. The combo-box on the top-right corner lists the available data types for the global variable.
4.1. COMMON MENUS

Only the standard data types that C++ directly supports are currently allowed (char, double, int, long, float, and short).

- The **Delete** button is used to delete the selected global variables.
- The **Close** button is used to hide the **Global Variables** editor dialog box.

**User-defined functions**

While building models, you might find it useful to define functions for the model using C++ header files and libraries. These functions are stored in C++ header (.h) or possibly source (.cpp) files that you must create. The files can be placed anywhere, but most often are stored in a new subdirectory in the project. If you are using C++ source files (.cpp), you will also need to place into the same directory a Makefile that will compile the source files into a library. The simplest way to create the Makefile is to copy one from an atomic model in your project and modify it to compile the library. With the current version of Möbius, you must compile the library manually before running a solver. Otherwise, an error message stating the library is missing will appear when the compiler tries to link with the user-defined library in the solver.

After you have successfully compiled the source code and created the library, you need to let Möbius know the location of the functions. You can do this by opening any model and selecting **USER-DEFINED HEADER** under the **EDIT** menu. A dialog box will pop up, asking for the location of your header (see Figure 4.2).

![Figure 4.2: User-defined header editor.](image)

Enter the path to the C++ header file. There is a check box that instructs Möbius to store the header file path relative to your Möbius project directory. You should check it if your header files are stored in a subdirectory of your project, as it makes it easier to share your project with other users, or relocate it to different systems. If the path entered is incorrect, or the file has not yet been created, Möbius will display an error message.
If you are also using a user-defined library, repeat the same operation for the library (see Figure 4.3). This dialog contains the same relative path checkbox found in the User-Defined Header dialog. It also contains another checkbox that makes it possible to support multiple operating system versions of the user-defined library. This feature will only work if the library name contains the name of the current operating system: Solaris, Linux, or Windows. After you check this box, the operating system name is replaced with $ARCH. When Möbius compiles your project on each operating system, $ARCH is replaced with the name of the appropriate operating system.

User-defined functions can be extremely useful if you are trying to make models more modular, or if multiple elements, such as SAN output gates, within the model are performing similar operations. In fact, you can even define functions that would modify the values of elements within the model (for example a place within a SAN). Here are a few examples in which user-defined functions are helpful:

- If your model contains complex functions that are repeatedly used or susceptible to change, you can write them in the library and simply call them from the model. For example, to define a comparison between two extended places that are arrays of 2 integers, you can create the following function in the library:

```c
inline int is_lower(int x1, int y1, int x2, int y2) {
    if (x1 < x2) return 1;
    else if (x1 == x2) {
        if (y1 <= y2) return 1;
        else return 0;
    }
    else return 0;
}
```
The following code shows how to use the function in the model:

```c
short result = is_lower(Place1->x, Place1->y,
                       Place2->x, Place2->y);
if (result == 1)
...
```

- In the same way, you can define enabling conditions for activities using user-defined functions:

```c
inline int activity_enable(int value1,
                           int value2) {
    int ret=0;
    if (value1==0 && value2==0)
        ret=1;
    ...
    return ret;
}
```

The enabling function of the corresponding input gate would then call the user-defined function, as follows:

```c
activity_enable(Command1->Mark(),
                Command2->Mark(), Group_ID->Mark(),
                Component_ID->Mark(),
                Component_type->Mark())
```

- If you have an extended place in your model with a type that is a large array, matrix, or structure, it is possible to write a user-defined function to initialize the extended place. If you want to create unique experiments that define different initial values for this extended place, you can create a single global variable, and then write the user-defined function so that the extended place is initialized with different values, based on the value of the global variable. Then in the study you only need to adjust one value to change the definition of all fields in the extended place.
Chapter 5

Atomic Formalisms

Each model is composed of one or more submodels, also referred to as atomic models. You can create and edit atomic models using different editors like the SAN editor, the PEPA editor, the Buckets and Balls editor, and the Fault Tree editor.

5.1 SAN

Möbius supports multiple formalisms, including stochastic activity networks (SANs). SANs \[24\] are stochastic extensions to Petri nets \[4\]. Using graphical primitives, SANs provide a high-level modeling formalism with which detailed performance, dependability, and performability models can be specified relatively easily.

5.1.1 SAN primitives

SANs consist of four primitive objects: places, activities, input gates, and output gates. Activities represent actions of the modeled system. Places represent the state of the modeled system. Input gates are used to control the “enabling” of activities, and output gates are used to change the state of the system when an activity “completes.”

**Places** Places represent the state of the modeled system. They are represented graphically as circles. In Figure 5.1 memory.chips, interface.chips, memory.failed, and computer.failed are places. Each place contains a certain number of tokens, which represents the marking of the place. The set of all place markings represents the marking of the stochastic activity network. Note that tokens in a place are homogeneous, in that only the number of tokens in a place is known; there is no identification of different kinds of tokens within a place.
The meaning of the marking of a place is arbitrary. For example, the number of tokens in a place could represent a number of objects, such as a number of tasks awaiting service. Also, the number of tokens in a place could represent an object of a certain type, such as a task with a certain priority level. This dual nature of a place marking provides a great deal of flexibility in modeling the dynamics of a system.

**Activities**  Activities represent actions in the modeled system that take some specified amount of time to complete. They are of two types: *timed* and *instantaneous*. Timed activities have durations that impact the performance of the modeled system, such as a packet transmission time or the time associated with a retransmission timer. Timed activities are represented graphically as thick vertical lines. In Figure 5.1, memory_chip_failure, and interface_chip_failure are timed activities. Each timed activity has an *activity time distribution function* associated with its duration. Activity time distribution functions can be generally distributed random variables. Each distribution can depend on the marking of the network. For example, one distribution parameter could be a constant multiplied by the marking of a certain place. Instantaneous activities represent actions that complete immediately when enabled in the system. They are represented graphically as thin vertical lines. No
instantaneous activities are represented in Figure 5.1. Case probabilities, represented graphically as circles on the right side of an activity, model uncertainty associated with the completion of an activity. Each case stands for a possible outcome, such as a routing choice in a network, or a failure mode in a faulty system. In Figure 5.1, activity interface chip failure has three cases. Each activity has a probability distribution, called the case distribution, associated with its cases. This distribution can depend on the marking of the network at the moment of completion of an activity. If no circles are shown on an activity, one case is assumed with a probability of one.

An activity is enabled when the predictes of all input gates connected to the activity are true, and all places connected to incoming arcs contain tokens, i.e. have non zero markings. Once enabled, the activity samples its delay distribution function to determine the time delay before the activity fires. When the activity fires it updates the state of the model by subtracting tokens from places connected by incoming arcs, adding tokens to places connected by outgoing arcs, and executing the functions in input and output gates. The specific order that the state updates occur is: input gates, input arcs, output gates, and finally output arcs.

Please be aware that the order in which state updates occur in Möbius differs from UltraSAN. Gates are executed before arcs in Möbius, but the opposite was true in UltraSAN. Also, if there are multiple instances of the same item, for instance multiple input gates, the order of application of the gates is not specified. Models must be constructed so that the gate functions are execution order independent anytime there are multiple instances of the same type of gates connected to an activity.

Also associated with each activity is a reactivation function. This function gives marking dependent conditions under which an activity is reactivated. Reactivation of an activated activity means that the activity is aborted and that a new activity time is immediately obtained from the activity time distribution. The reactivation function consists of an activation predicate and a reactivation predicate. An activity will be reactivated at the moment of a marking change if (1) the reactivation predicate holds for the new marking, (2) the activity remains enabled, and (3) the activation predicate holds for the marking in which the activity was originally activated.

Input gates Input gates control the enabling of activities and define the marking changes that will occur when an activity completes. They are represented graphically as triangles. In Figure 5.1 IG1 and IG2 are input gates. On the other side of the triangle is a set of arcs to the places upon which the gate depends, also called input places. Each input gate is defined with an enabling predicate and a function.
The enabling predicate is a Boolean function that controls whether the connected activity is enabled. It can be any function of the markings of the input places. The input gate function defines the marking changes that occur when the activity completes.

If a place is directly connected to an activity with an arc, it is equivalent to an input gate with a predicate that enables the activity when ever the place has more than zero tokens along with a function that decrements the marking of the place when ever the activity fires.

**Output gates** Like input gates, output gates define the marking changes that will occur when activities complete. The only difference is that an output gate is associated with a single case. An output gate is represented graphically as a triangle with its flat side connected to an activity or a case. In Figure 5.1, OG1, OG2,..., OG6, and OG7 are output gates. On the other side of the triangle is a set of arcs to the places affected by the marking changes. An output gate is defined only with a function. The function defines the marking changes that occur when the activity completes. There is also a default scenario for output gates. If an activity is directly connected to a place, it is equivalent to an activation in which an output gate has a function that increments the marking of the place whenever the activity is fired.

More information on SANs can be found in [24]. The next few sections describe the various features available in the SAN editor to develop a SAN model.

### 5.1.2 Editor

This section looks into the atomic formalism that represents stochastic activity networks with emphasis on creation, editing, and manipulation of atomic models using the Möbius SAN editor. Refer to Chapter 3 for details on how to create and open a SAN model.

The names of the selected project and subnet appear in the top left corner (see Figure 5.1). The large gridded area in the center is the drawing area. In the case of a new model, this gridded area is blank. Like most applications, the editor lists the menu horizontally at the top of the application’s current active window. If you click on a menu item, it drops a tool panel containing several options. The menu items and tool panel options are used to create and draw in the SAN formalism, as discussed below.

As discussed in Section 4.1, many menus are common to all model editors within Möbius. Please see Section 4.1 for detailed descriptions of the common editor functions. The following paragraphs will discuss the functionality that is unique to the SAN editor.
5.1.3 **Edit**

**Type definitions**

The SAN formalism supports a special element called *extended place* that allows the model to handle the representation of *structures* and *arrays* of primitive data types. Clicking on TYPE EDITOR opens the dialog box shown in Figure 5.2. In this dialog box, you can create new structures and arrays that can be associated with extended places.

- Use the **Edit** button to edit any currently defined user-defined types. The GUI can automatically tell if the user is editing an array (see Figure 5.4) or structure (see Figure 5.3) and appropriately open the required editor. You are expected to click on the available list of user-defined types to edit them. The editor allows you to change the name of any user-defined type, even if the user-defined type is declared as a field in another user-defined type or associated with an extended place. The interface takes care of updating all the changes.

- Use the **Delete** button to delete the selected user-defined type that is not being used by any other user-defined type and is not associated with any extended place. If you attempt to delete a type that is in use, an error dialog box pops up. A successful delete removes the user-defined type from the current list.

- Use the **New Struct** button to pop up a new dialog box that allows you to create a new structure-type definition. The pop-up dialog box layout is depicted in Figure 5.3.
Figure 5.3: SAN struct-type definition editor.

Figure 5.4: SAN array-type definition editor.
5.1. SAN

- Use the Name text-box to define a name for the structure.
- The Fields list shows the fields you declared in the current structure.
- The New field button adds a new field to the structure. The combo-box on the left lets you choose the type of the field from a list of the standard data types and user-defined data types. Recursive definitions or pointers are not allowed.
- The Delete button deletes the selected field.
- The Move up button moves the selected field up by one position.
- The Move down button moves the selected field down by one position.
- The OK button accepts all changes made and hides this dialog box, bringing the User-type definitions editor to the front. An error message pops up if you try to create a structure with no fields or a structure that has no-name or a non-unique name.
- The Cancel button discards all changes, hides this dialog box and returns you to the User-type definitions editor.

- The New Array button pops up a new dialog box that allows you to create a new array-type definition. The pop-up dialog box layout is depicted in Figure 5.4.
  - The Name text-box allows you to define a name for the array.
  - The Definition combo-box allows you to select the data type for the array. The choices include primitive data types or user-defined types. You are also expected to declare the size of the array. The elements of the array are generically labeled as index.
  - The Initialize all array elements to common value checkbox makes it easier to manage large arrays. If it is known when the array type is defined that all elements in the array should be initialized to the same value, 0 for instance, clicking this box will avoid the need to initialize each element separately. This option is critical when defining very large arrays because it saves a significant amount of memory within the Mobius editor process.
  - The OK button accepts all changes made, hides this dialog box, and returns you to the User-type definition editor. An error message will pop up if you try to create an array with an invalid size (i.e., a non-numeric, zero, or negative size) or an array that has no name or a non-unique name.
5.1.4 View

Increase size and decrease size

Click on menu item INCREASE SIZE to increase the size of the drawing area. Click on DECREASE SIZE to decrease the size until the scroll bar disappears or becomes equal to area on which the model elements are spread. You can increase the size of the drawing area, scroll down to the extra space created, and add new components to the model, if necessary.

Zoom in and Zoom out

Each time you click on the menu item ZOOM IN, the GUI enlarges the drawing area by a fixed percentage. The interface takes care of adding scroll bars if necessary so that the entire model can be viewed. To keep the model in focus, be careful not to multiple-click on this option. ZOOM OUT decreases the size of the model until the scroll bars disappear.

Grid Setting

Click on the menu item GRID SETTING to open a dialog box as shown in Figure 5.5. Here, you can define the X and Y coordinate size of the grid in pixels. Note that if you set the grid to 1 pixel size, it will take an extremely long time to
render the model. You can adjust the grid according to your comfort level. There are a few check-boxes on the Grid Setting dialog box. They are Enable display grid points, Enable snap to grid for object placement, and Snap existing objects to grid. Each check-box is self-explanatory. Use the OK button to accept the changes or the Cancel button to discard the changes. Either way the dialog box will close so that you are returned to the SAN Editor.

5.1.5 Elements

Elements are SAN model primitives. The ELEMENTS menu includes the following types of elements:

- **PLACE**, which will be represented by a blue circle.
- **EXTENDED PLACE**, which will be represented by an orange circle.
- **INPUT GATE**, which will be represented by a triangle with its tip pointing to the left.
- **OUTPUT GATE**, which will be represented by a triangle with its tip pointing to the right.
- **INSTANTANEOUS ACTIVITY**, which will be represented by a solid vertical bar. If the activity has multiple cases, they appear as small circles on the right side of the vertical bar.
- **TIMED ACTIVITY**, which will be represented by a hollow vertical bar. It has small circles on its right side if it has multiple cases.

You can also select a SAN model element by clicking the left button on the required component while the cursor is on the menu item ELEMENTS. SAN elements can also be accessed via icons below the menu. To place one of those components, click on it, move it to the desired location, and place it by clicking the left mouse button. A dialog box will appear asking you to specify various attributes relevant to the element type. The name must be unique across the current atomic model. The name also must follow the C++ variable naming convention. The rest of this section discusses various parameters of each model element.

**Place**

When you create or edit a place (by right-clicking on the place and clicking on EDIT in the pop-up window), a dialog box similar to the one shown in Figure 5.6 appears.
• The **Name** text-box allows you to modify the name, provided that the new name is unique in the current model.

• The **Token(s)** text-box allows you to edit or modify the initial marking. The value should be a non-negative short integer, which implies that place markings can only be non-negative integers less than or equal to 32,767. However, the initial marking may be defined as a scalar value, or as a global variable. The following points about global initial marking must be noted:

  – global variables for places must be of type short.
  – global variables for initial markings should not be arithmetic expressions, or functions of other initial global markings.

• The **OK** button accepts the changes and returns you to the **SAN Editor**. It pops up an error message if there is a problem in validating the tokens text-box.

• The **Cancel** button discards the changes and brings you back to the **SAN Editor**.

**Extended place**

When you create or edit an extended place (by right-clicking on an extended place and clicking on **EDIT** in the pop-up window), a dialog box similar to that shown in [Figure 5.7](#) appears.

• The **Name** text-box allows you to modify the name provided that the new name is unique in the current model.
Figure 5.7: Extended Place Attributes dialog box.
• The initial value tree represents the current values associated with the structure or array of the extended place. Only the leaf nodes can be initialized to a particular initial value. Clicking on intermediate nodes hides the visible text-box. Each time you click on a new leaf node, the GUI evaluates the current initial value for validity, displaying error messages if necessary.

• The text-box allows you to edit/modify the initial marking(s) of the selected leaf nodes of the structure or an array. However, the initial marking may be defined as a scalar value, or as a global variable. The following points about global initial marking should be noted:
  – global variables for extended places must be of the same type as the corresponding leaf field.
  – global variables for initial markings should not be arithmetic expressions, or functions of other initial global markings. Note that the current SAN editor only allows global variables of primitive data types.

• The type combo-box allows you to associate a user-defined type with an extended place. Whenever you change the user-defined type, its values are initialized to 0. The Tree is re-drawn with the new structure or array.

• The OK button accepts the changes and brings you back to the SAN editor. It pops up an error message if there is a problem in validating the tokens text-box.

• The Cancel button discards the changes and returns you to the SAN editor.

Input gate

When you create or edit an input gate (by right-clicking on a place and clicking on EDIT in the pop-up window), a dialog box similar to the one shown in Figure 5.8 is displayed.

When defining an input gate, be aware of the following points:

• You must specify a predicate and a function.

• Input predicates must return a Boolean value. They may be an expression or a sequence of C statements.

• Input functions are sequences of C++ statements.

• No return statements are needed in function specifications, since their action is to change the marking of a SAN, not to return a value.
5.1. SAN

- If no action is desired in a function (the identity function), this is specified by a lone semicolon.

- Global variables, places, and extended places may be used in defining input gates.

**Special note on programming with extended places**  While defining the function of an input gate or output gate, you can use extended places. You can access the fields of a structure using standard C++ dereferencing of pointers. Consider a scenario in which an extended place “ep” is associated with a user-defined structure called “coordinates” as shown in Figure 5.8, with three fields “x,” “y,” and “id” of types `short`, `short`, and `char`. To access field “x” in any function of a
gate, use \texttt{ep->x->Mark()}. In another scenario, extended place “ep” is associated with a user-defined array called “ten_type” that is an array of ten elements of type \textit{short}. Each of these elements of “ep” can be accessed using the method \texttt{ep->Index(i)->Mark()} where “i” can be a previously defined variable. Remember, it is possible to program user-defined variables with arrays and structures using a similar technique by applying the above-discussed rules recursively.

\textbf{Output gate}

When you create or edit an output gate (by right-clicking on a output gate and clicking on \textit{EDIT} in the pop-up window), a dialog box similar to that shown in Figure 5.9 appears.

Be aware of the following points regarding definition of output gates:

- Only one function is needed.
- Output functions are sequences of C++ statements.
• No return statements are needed in function specifications, since their action is to change the marking of the SAN, not to return a value.

• If no action is desired in a function (the identity function), this is specified by a lone semicolon.

• You may use global variables, places and extended places while defining output gates. Refer to comments in the previous section on programming with extended places.

Activity

The SAN formalism models activities with two types of elements: timed ac-
tivities and instantaneous activity. Right-clicking on an activity element pops up a menu that has an option to edit the activity. A dialog box similar to those shown in Figure 5.10 and 5.11 may appear depending on the type of activity created. The steps involved in defining the activities of the model element timed activity are as follows:

- Specify an activity distribution function by clicking the selector next to the desired distribution in the box titled Time distribution function. For analytic solutions, only exponential and deterministic distributions may be selected.

- Next, enter the parameters for the chosen distribution in the box below the combo-box. The headings on the box change depending on the distribution chosen. For the exponential distribution, only the rate, which is the reciprocal of the mean, is required. Section 1.2.5 describes the distribution functions for timed activities in Möbius together with their parameters. The parameters can be expressions (double-precision values), global variables of type double, or series of statements. If statements are used, you must provide a return statement with a double-precision value equaling the parameter.

If more than one case is specified, an additional sub-window is present for specifying the case probabilities. Only one case is visible at a time, but you can step through the remaining cases by clicking on the tab of the case number. Some points to be noted are:

- The first case probability corresponds to the top circle on the activity.

- Case probabilities can be expressions (double-precision values between zero and one), or series of statements (as in Figure 5.10). If a case probability is a series of statements, then you must provide a return statement with the desired value.

- Case probabilities may depend on the markings of places, but need not be connected to those places.

- Case probabilities can be functions of global variables, place markings, and extended place markings.

- Möbius automatically normalizes the case probabilities.
5.1. SAN

**Execution Policy**

The execution policy editor is an advanced concept that only applies to simulation-based solutions and can be safely ignored for many models. Reactivation is disabled by default, and often this behavior is sufficient. Reactivation is disabled when both the activation and reactivation predicates are undefined, or have a defined value that is false. Refer to section 1.2.4 for more information on execution policy.

Reactivation is used to change the behavior of the activity during simulation. When reaction is disabled, the distribution function of an activity is sampled only during the state where the activity is first enabled (i.e., when all input gates associated with the activity are enabled. In more advanced scenarios, the behavior of the model might require that the activity might be activated again (re-activated) so that its distribution is resampled based on the new state of the model. In such cases, the user can set predicates in the Activation Predicate and Reactivation Predicate boxes. In each state where the activation predicate is true, the simulator will check the given activity to determine if it should be reactivated. If the reactivation predicate is also true in the current state, the activity will be reactivated, meaning a new delay will be sampled from the distribution function, and the activity will be scheduled to fire after “delay” units of time from the current simulation time. The reactivation predicate affects the activity only if the activity is already enabled (Figure 1.3(d)). Enabling of the activation predicate could possibly cause scenario (c) in Figure 1.3.

Figure 5.12 depicts the dialog box that appears with the Execution Policy editor.

**Arcs and lines**

In SAN models, three arc tools are used for drawing: straight, spline, and connected lines. You use them by selecting one of the arc tools with the left mouse button. Move the mouse to the desired point of origin of the line, and click the left button. Move the mouse to the desired locations of all intermediate points one by one, clicking the left mouse button (for splines and connected lines) to set down each point. When the destination is reached, click the left button. If one of the end points of the line cannot be identified as an appropriate object, an error message appears. You can avoid that by placing the end points clearly within the boundaries of the desired endpoint objects. Lines may overlap objects or other lines. You can cancel on arc you are drawing by right-clicking the mouse.

Note that lines are directed and thus may be drawn only between certain objects in a certain order as listed below:
CHAPTER 5. ATOMIC FORMALISMS

Figure 5.12: Execution Policy editor.

- from a place to an activity
- from a place or an extended place to an input gate
- from an activity to a place
- from an activity to an output gate
- from an input gate to an activity
- from an output gate to a place or an extended place

Any other combination or order causes an error message to appear. An arrow is automatically placed at the head of the arc when you connect a place directly to an activity, or an activity directly to a place.

You do not need to draw arcs between input/output gates and the places/extended places. Doing so aids in keeping track of interaction between these elements. However, the arc can be left out if including it would clutter the model.

**Text boxes**

Text boxes have been provided to allow programmers to write comments about the model in the drawing area. They do not affect the behavior of the model in any
way. Text boxes are meant to be useful tools for documenting parts of the model. To enter a text box within the drawing area, first select the TEXT BOX element from the ELEMENT menu. Next, place the cursor (which appears as an arrow) by the object in the drawing area by which the text is to be entered and click on the left button. Write the text to be displayed in the pop-up dialog that appears. You can edit the text by right-clicking on the text box and selecting EDIT.

5.1.6 Useful techniques

This section outlines tricks and techniques that advanced users might find helpful when constructing their models.

Simulation time

If you would like for components in the model to access the current simulation clock, its value can be accessed from C++ code within a given SAN component using this static class member: BaseModelClass::LastActionTime. The definition of BaseModelClass can be found in <Mobius Installation Root>/Cpp/BaseClasses/BaseModelClass.h.

The value is usable only when the model is solved via simulation; thus, models that depend on this value cannot be solved using the state-space-based numerical solution techniques available in Möbius.

Direct use of distribution functions

Advanced users have asked for the ability to generate random numbers using the same set of distribution functions that are used by Möbius. They can do so within the C++ code of SAN model components by adding code similar to this to the component definition:

```
static UserDistributions dist(0, 31415);
double delay=dist.Exponential(.5);
```

The first parameter passed to the UserDistributions constructor specifies the type of random number generator to use with this UserDistributions object (0 is Lagged Fibonacci; 1 is Tausworthe). The second parameter sets the initial seed of the random number generator (31415 is known to provide a relatively long period).

For a complete list of the available distribution functions and their arguments, please refer to this Möbius AFI header file:

```
<Mobius Installation Root>/Cpp/simulator/Distributions.h
```
5.2 Buckets and Balls

Another formalism supported by Möbius is that of Buckets and Balls. Buckets and Balls provides a method for modeling simple systems for which a SAN might not be necessary. While it lacks the capabilities of a SAN, it is often more appropriate for those systems and subsystems that do not require the complexity offered by other formalisms.

5.2.1 Buckets and Balls primitives

Buckets and Balls consists of two primitive objects: buckets and transitions. Buckets represent the current state of the system, and transitions represent activities and are used to change the state of the system.

**Buckets**  Buckets are used to represent the state of the model. In the editor, they are represented graphically as circles. Each bucket can contain a certain number of balls, which are used to indicate system state. All balls in a given bucket are assumed to be homogeneous. Thus only the current number of balls in a given bucket is known; there is no method for distinguishing one ball from another in a given bucket.

As with SAN models, the meaning of the state of a given bucket is arbitrary. It could just as easily be jobs in a queue, failures propagating through a system, or an inventory model, for example.

**Transitions**  Transitions represent events that can change the state of the system. There are two types of transitions available for the Buckets and Balls formalisms: timed and instantaneous. Timed transitions are fired according to their time distribution functions. Transition time distribution functions can be any generally distributed random variables and can be dependent on both variables and the current state of the system. All transitions are represented graphically as lines with arrows indicating the direction of the transition. Timed transitions fire at their generated event times. Instantaneous transitions complete immediately when enabled by the system. It is important to note that only one instantaneous transition may fire from a given bucket at any one time. As there are often many transitions from a single bucket, it is necessary to determine an order for the firings. This order is determined based on the transitions’ weights and ranks. For each rank, all transitions are considered and identified as valid or invalid; the order in which the valid transitions are fired is determined probabilistically. If no valid transitions exist of a given rank, those of a lower rank are considered in the same manner. After a timed transition fires, instantaneous transitions will continue to fire until no more
are valid, at which point the model will activate the next timed transition on the event list and proceed as before.

5.2.2 Editor

This section covers the atomic formalism of Buckets and Balls including the creation, editing, and manipulation of atomic models using the Möbius Buckets and Balls editor. Refer to Chapter 3 for details on how to create and open a Buckets and Balls model.

![Figure 5.13: Buckets and Balls editor.](image)

As in the SAN model editor, the name of the selected project appears in the top left corner along with the name of the current atomic model. The grid in the center of the interface is the drawing area and is initially blank for a newly created model. Drop-down menus are listed horizontally at the top of the active window and contain tools and options for working with the Buckets and Balls formalism. Some of the menus are common to all model editors in Möbius; please see Chapter 4.1 for a detailed description of these common editor functions.

5.2.3 Edit

The Buckets and Balls formalism has no unique features in the Edit menu; all functions are the same as those described in Chapter 4.1.

5.2.4 View

The view menu for the Buckets and Balls formalism is identical to that found in the SAN formalism. Please refer to Chapter 5.1.4 for more information on the options provided in this menu.
5.2.5 Elements

Elements are Buckets and Balls model primitives. The ELEMENTS menu includes the following types of elements:

- **BUCKET**, which will be represented by a blue circle.

- **TIMED TRANSITION**, which will be represented by a black line with an attached arrow indicating the direction of the transition.

- **INSTANTANEOUS TRANSITION**, which will also be represented by a black line with an attached arrow indicating the direction of the transition.

Buckets and Balls model elements can be selected either through the menu item ELEMENTS or via the icons below the menu. To place one of these components, simply click on it, move it to the desired location, and place it by clicking the left mouse button. A dialog box will then appear, requesting that you specify the attributes related to the element’s type. As with the SAN model formalism, all names must be unique across the atomic model. All names must also conform to C++ variable naming conventions. For more information on placing transitions, please see the subsection on arcs and lines below.

**Buckets**

When you are creating or editing a bucket (which you do by right-clicking on the Bucket and selecting **EDIT** in the pop-up window) a dialog box similar to the one illustrated in Figure 5.14 is shown.

- The **Name** text box allows you to modify the name of the bucket, provided the new name is still unique in the current model.

- The **Ball(s)** section of the dialog contains a radio button to allow you to select either a constant or variable to represent the number of balls in this bucket.

  - If **Constant** is selected, you must enter a non-negative short integer, that is any number greater than or equal to zero and less than or equal to 32,767.

  - If **Variable** is selected, you must enter a global variable of the type short.
5.2. BUCKETS AND BALLS

Figure 5.14: Buckets and Balls bucket editor

- The OK button accepts any changes you may have made and returns you to the Buckets and Balls Editor. If there are any problems with changes you have made, an error message will appear, indicating the error and asking you to rectify it.

- The Cancel button discards all changes and returns you to the Buckets and Balls Editor.

Timed Transition

When you are creating or editing a timed transition (which you do by right-clicking on a timed transition and clicking on EDIT in the pop-up window), a dialog box similar to that illustrated in Figure 5.15 is shown.

- The Name text box allows you to modify the name, provided that the new name is unique in the current model.

- The Arc Cardinality text box allows you to set the number of balls transferred by the timed transition when it fires. Note: the transition will not occur for a given bucket if the number of balls in the bucket is less than the number set in this text box.

- The Time distribution function section of the dialog contains a pull-down menu of the general distributions available for a timed transition and a Parameters button that opens a dialog to allow you to parameterize the selected
Figure 5.15: Buckets and Balls timed transition editor.
distribution. Section 1.2.5 describes the distribution functions for timed activities in Möbius along with their parameters. The parameters can be expressions, global variables of type double, or a series of statements. As with the SAN atomic models, if a series of statements is used, you must provide a return statement with a double-precision value for the parameter.

- The **OK** button accepts any changes you may have made and returns you to the Buckets and Balls Editor. If there are any problems with changes you have made, an error message will appear, indicating the error and asking you to rectify it.

- The **Cancel** button discards all changes and returns you to the Buckets and Balls Editor.

**Instantaneous Transition**

When you are creating or editing an Instantaneous Transition (which you do by right-clicking on an instantaneous transition and clicking on **EDIT** in the pop-up window), a dialog box similar to that illustrated in Figure 5.16 is shown.

![Instantaneous transition attributes](image)

Figure 5.16: Balls and Buckets instantaneous transition editor.

- The **Name** text box allows you to modify the name, provided that the new name is unique in the current model.
• The Arc Cardinality text box allows you to set the number of balls transferred by the timed transition when it fires. Note: the transition will not occur if the number of balls is less than the number listed in the text box.

• The Weight text box allows you to set a probabilistic weight for the transition (as only one transition may be selected out of each bucket), these weights will be normalized for all transitions of the specified rank for a given bucket. The weight may be expressed as a variable or as a double-precision value.

• The Rank text box allows you to set the rank of this transition, indicating its priority with respect to the bucket of origin. Higher-valued ranks will be selected first unless they cannot fire, in which case lower-valued ranks will be selected. Rank may be expressed as a variable or a constant.

• The OK button accepts any changes you may have made and returns you to the Buckets and Balls Editor. If there are any problems with changes you have made, an error message will appear indicating the error and asking you to rectify it.

• The Cancel button discards all changes and returns you to the Buckets and Balls Editor.

**ArCs and Lines**

In Buckets and Balls models, as in SAN models, there are three arc drawing tools at your disposal: *straight*, *spline*, and *connected* lines. In a Buckets and Balls model, these arcs define the timed and instantaneous transitions. In order to draw an arc, you simply select one of the arc tools with the left mouse button. Move the mouse to a bucket and click with the left mouse button to set the origin bucket for the transition. If you are drawing a spline or connected line, click the left mouse button in the drawing area to set control points. Finally, click the left mouse button on the destination bucket to create the transition arc. If at any point you wish to cancel the arc, right-click on the drawing area. ArCs may overlap other objects and other arcs. All arCs and lines are directed, and therefore must have both of their endpoints anchored at buckets.

**Text Boxes**

As in the SAN model editor, text boxes have been provided to allow programmers to write comments about different aspects of their models within the drawing area. These comments have no effect whatsoever on the functioning of the model,
although they may help another programmer understand the model. For more information on creating and editing text boxes, please see Chapter 5.1.5.

5.3 PEPA

Möbius also supports PEPA as a modeling formalism. PEPA, unlike the SAN or the Buckets and Balls formalisms, is a process algebra. Instead of creating a PEPA atomic model in a graphical editor, the model is created using an editor provided in Möbius in a form that resembles a formal language. Möbius extends the basic PEPA language to include process parameters, guards, and value passing. This extension is known as the PEPA$_k$ language.

5.3.1 The PEPA$_k$ language

This section covers the basic building blocks of the PEPA$_k$ language including process variables, formal parameters, guards, choices, prefixes, hide, and cooperation.

- **Process Variable**: In the PEPA$_k$ language, processes are represented using process variables. A process variable is defined by assigning an equation to a symbolic name using the assignment operator “=”. For example the following defines the process variable $P$:

\[
P[a] = [a > 5] => (\alpha, x).P[a - 1]
+ [(a > 0) & &(a < 5)] => (\beta, x).P[a - 1];
\]

- **Formal Parameters**: The PEPA$_k$ language extends PEPA to include the possibility of formal parameters for process variables. In the example above the process variable $P$ takes a single parameter, $a$. More parameters may be specified by a comma delimited list as follows: $P[a, b]$.

- **Guards**: The introduction of guards to the PEPA$_k$ language allows the selective enabling and disabling of a process expression based on the guard conditions. A process expression is enabled if its guard evaluates to true, and disabled if its guard evaluates to false. Guards are represented by boolean expressions enclosed in square brackets before a process expression, and may
contain references to formal parameters. In the above example, the two process expressions are guarded by conditions, \([a > 5]\), and \([((a > 0)\&\&(a > 5))\]. Thus the first process expression is enabled only if the parameter \(a\) is greater than five, and the second process expression is enabled only if the parameter \(a\) is greater than zero and less than five. The guard operator “\(=>\)” assigns a process expression to a given guard expression.

- **Choice**: The choice operator “\(+\)” is used to allow a process to behave in different ways under different circumstances as defined by a process expression. The first enabled process expression to complete is selected and the remainder of the expressions are discarded. In the above example we see a choice between two guarded process expressions.

- **Prefix**: The simplest way to describe the behavior of a process is to add a prefix to the references to the process variable. Prefixes are denoted using the form \((\alpha,x).P\), where \(\alpha\) is some action which has a duration that is exponentially distributed with rate \(x\). After the action has completed, it then behaves according to the referenced process variable. In the case of a choice between multiple enabled processes expressions, it is the activity that completes first which is selected.

- **Hide**: The hide operator “\(/\)” is a combinator that is used to hide activities. With respect to a process, the hidden activities are considered silent. As in many other process algebras, the combinator is used to express abstraction.

- **Cooperation**: Processes \(P\) and \(Q\) are said to cooperate over a set of shared activities if they contain one or more shared activities and are defined as cooperating over these activities. They are defined as cooperating over an activity through the definition of a new process variable as follows:

\[
S = P[x] < \alpha > Q[y]
\]

The “\(<\)” operator is the cooperation operator and takes its parameters as a comma delimited list of activities to be shared between the cooperating processes. If a process enables an activity within the shared set, it will not be able to continue with the activity until the cooperating process also enables this activity. Both processes then complete the shared activity. When defining a prefix for a shared activity, one of the cooperating processes may define it with rate \(T\), indicating that it is a passive activity and the duration of the activity is to be determined based on the duration generated by the other
5.4 Fault Trees

Möbius also supports a fault tree formalism for analyzing system reliability. Although originally developed in the 1960’s to model the unreliability of the Minuteman missile system, fault trees have been extended to model computer systems as well [23]. Component failures are modelled as leaves of a tree, with system failure as the root. Component failures are connected by a series of logic gates representing the relationships between failures in the real system. By doing so, we can analyze the relationships between component failures, and the total system failure as well.

Typically, fault trees can be categorized as either static or dynamic. Static fault trees use logic gates to establish relationships between component failures, but suffer from the fact that they cannot model sequential relationships between failures. Dynamic fault trees, on the other hand, can model sequential failures between components [22]. In Möbius, the fault tree formalism includes all of the features of static fault trees, but only a subset of the features of dynamic fault trees. An example of a fault tree that uses both static and dynamic components is in Figure 5.17.
5.4.1 Fault tree primitives

Fault trees consist of three primitives: nodes, events, and logic gates. Nodes represent the highest level performance we are trying to analyze, or the highest level of any submodel we are trying to analyze. Events describe when failures occur and how they affect others, while logic gates are used to connect failure events in a logical manner.

Nodes Nodes are analogous to the places in SANs. In fault trees, nodes represent failure states of the modelled system, whether at the highest level (the root) or at any intermediate level of the tree. In either case, a node with value greater than zero represents a submodel that has failed.

Events Events represent failures of components in a system. They are similar to timed activities in SANs and support the distributions described in Section 1.2.5.

Logic Gates Logic gates perform the work in a fault tree. They represent the connections between component failures and describe how submodels interact. Logic gates fall into two categories: static or dynamic. The fault tree formalism supports...
the And, Or, Xor, and K-of-N static logic gates and the Priority AND dynamic logic gate.

5.4.2 Editor

This section covers the fault tree editor, including the creating and modification of fault tree primitives. Most of the editor is the same as for SANs, so refer to Chapter 4 for information on how to create and open fault tree models and Section 4.1 for details on common editor features.

5.4.3 Edit

The fault tree formalism has no unique features in the EDIT menu. All functions are the same as those described in Section 4.1.

5.4.4 View

The fault tree formalism has no unique features in the VIEW menu. All functions are the same as those described in Section 4.1.

5.4.5 Element

Elements are fault tree model primitives. They include the common primitives of cursor, text block, straight connection, connected line, and spline curve. More information about the connections, lines, and text boxes can be found in Sections 5.4.5 and 5.4.5.

In addition, they include the following fault-tree-specific elements:

- **Node**, represented by smaller blue circles.
- **Event**, represented by larger red circles.
- **And Gate**, represented by a blue AND symbol.
- **Or Gate**, represented by a blue OR symbol.
- **Xor Gate**, represented by a blue XOR symbol.
- **K-of-N Gate**, represented by a blue XOR symbol, with a white number in the middle representing the value of K.
- **Priority AND Gate**, represented by a blue Priority AND symbol.
Fault tree model elements can be selected from the ELEMENTS menu or by clicking on the appropriate icon. In either case, the element is placed on the model by clicking the left mouse button. After doing so, an element-specific dialog will appear to enable modification of the default parameters. As in all Möbius atomic formalisms, names must be globally unique.

**Basic Static Logic Gates**

The basic logic gates including *and*, *or*, and *xor* have a common dialog box similar to Figure 5.18.

- The Name text box allows you to modify the name, provided the new name is globally unique in the model. An error box will appear if it is not.

- The Ok button accepts changes to the element and returns to the Fault Tree editor. If there are any errors, an error message will appear.

- The Cancel button disregards any changes you have made to the element and returns you to the Fault Tree editor.

**Advanced Static Logic Gates**

In addition to the basic static logic gates in Section 5.4.5, the fault tree atomic formalism supports the K-of-N logic gate. For K-of-N logic gates, the output of the gate is active only if at least K of its inputs are active. For example, a four-input K-of-N gate with K = 2 would only be active if at least half of its inputs were active. The dialog to modify a K-of-N gate is shown in Figure 5.19.

- The Name text box allows you to modify the name, provided the new name is globally unique in the model. An error box will appear if it is not.

- The K Value text box allows you to modify the value of K for the logic gate. Acceptable values range from 0 to N, where N is the number of inputs to the gate.
5.4. FAULT TREES

The Cancel button disregards any changes you have made to the element and returns you to the fault tree editor.

Dynamic Logic Gates

The Möbius fault tree formalism supports the Priority AND logic gate. With a Priority AND gate, the output is only active if the inputs become active in a particular order defined by the user. This order can be changed by the dialog box in Figure 5.20.

- The Name text box allows you to modify the name, provided the new name is globally unique in the model. An error box will appear if it is not.
- The list box on the left contains the current inputs to the Priority AND gate. Elements higher in the list have higher priority.
• The **Up** button moves the currently selected input up the priority list.

• The **Down** button moves the currently selected input down the priority list.

• The **Ok** button accepts changes to the element and returns to the fault tree editor. The order of the inputs from top to button in the listbox are now the priority order for the Priority AND logic gate. If there are any errors, an error message will appear.

• The **Cancel** button disregards any changes you have made to the element and returns you to the fault tree editor.

**Arcs and Lines**

Like the SAN (and other) formalisms, the fault tree formalism gives three drawing tools to connect logic gates, events, and nodes. They include *straight connection*, *connected line*, and *spline curve*. The details of how arcs and lines can be placed and modified is described in Section 5.1.5.

Lines and arcs in the Möbius fault tree formalism can only be drawn between objects in certain orders:

• from a logic gate to any other logic gate

• from a logic gate to a node

• from an event to a node

• from an event to any logic gate

**Text Boxes**

Text boxes in the fault tree formalism are identical to those in SANs. Please refer to Section 5.1.5.

### 5.5 External Atomic Interface

The external atomic formalism is included in the current release of Möbius and will be documented in a future edition of the manual.
Chapter 6

Composition Formalisms

The Möbius tool allows for the construction of composed models from previously defined models. This gives the modeler the ability to adopt a hierarchical approach to modeling, by constructing submodels as meaningful units and then combining them together in a well-defined manner to construct a model of a larger system. 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.

The composition method supported in Möbius is the state-sharing approach. In this approach, submodels are linked together through superimposing (i.e., sharing of corresponding state variables, including no more than one state variable from each subset) of a subset of the state variables of each submodel. This allows submodels to interact in such a way that each of them can read from and write to the shared state variables. Any write to a shared state variable will consequently be available to all submodels that contain that state variable. For example, it is possible to compose two SAN models by having them hold a particular place in common.

Notice that Möbius models are closed under composition. This means that a composed model is also a model itself, and therefore can be further composed with other submodels to produce larger models. Although a composed model is a single model with its own state space, it is not a “flat” model. It is hierarchically built from submodels, which largely preserve their formalism-specific characteristics, so the composed model does not destroy the structural properties of the submodels. Moreover, the compositional techniques do not depend on the particular formalisms of the submodels that are being composed, provided that any requirements are met.

Currently, Möbius features two state-sharing composed model formalisms: Replicate/Join composition and Graph composition.
6.1 Replicate/Join

This section begins with an overview of the Replicate/Join composed model formalism as well as definitions of terms that will be used throughout the section. Then, the different parts of the Replicate/Join composed model formalism and the way they can be used to build a composed model will be described.

6.1.1 Overview

The Replicate/Join composed model formalism was originally conceived for SAN models [30]. However, in the Möbius tool, it can be used with any atomic or composed formalism. The formalism enables the modeler to define a composed model in the form of a tree, in which each leaf node is a predefined submodel node (atomic or composed), and each non-leaf node is classified as either a Join node or a Replicate node. The root of the tree represents the complete composed model.

A Join is a general state-sharing composition node used to compose two or more submodels. A Join node may have other Joins, Replicates, or other submodels defined as its children.

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

Since the instances of a Replicate composed model are indistinguishable, Möbius is able to exploit this structural symmetry and generate a lumped state space. The lumped state space is smaller than the unlumped state space in which symmetries are not present or exploited. Thus, when appropriate, use of a Replicate node instead of a Join node can lead to less time- and space-consuming state-space-based numerical solution. Details of the symmetry-based optimization remain private inside a Replicate/Join model, and the rest of the Möbius tool has no need to know the details of such optimizations. A more detailed description of this optimization is available in [30].

6.1.2 Replicate/Join composed model editor

Figure [6.1] shows the Replicate/Join composed model editor. As with other model editors, this editor can be opened either by creating a new model or by opening
an existing one. To create a new Replicate/Join, select Composed or any of its children in the project window and select the NEW command either by choosing it in the context menu or by clicking the leftmost icon in the toolbar (see Figure 3.1). To open an existing Replicate/Join model, select the OPEN command either by choosing it in the context menu or by clicking the corresponding icon.

This model editor is divided into the following sections:

- Top menu bar. It consists of the following five menus: FILE, EDIT, VIEW, ELEMENTS, and HELP.
- Toolbar. It gives the user one-click access to some of the menu commands.
- Editor pane. The composed model is drawn in this area.
- Status pane. The version of Möbius running and the version of the composed model that is open are shown in this pane.

FILE, EDIT, VIEW, and HELP menus have sets of commands similar to those in the previously described model editors. For more information on those menus, refer to Section 4.1. The menus unique to the ELEMENTS menu in this editor are explained below.

**ELEMENTS menu**

As with other GUI-based model editors, the ELEMENTS menu (shown in Figure 6.2) contains the set of all elements from which a Replicate/Join composed
model can be built. The order of the elements from top to bottom is the same as the order of buttons on the toolbar.

- **CURSOR** is the first (starting from left) icon in the toolbar, used to select elements in the editor pane.

- **REP** (abbreviation for Replicate) and **JOIN** are unique to this editor and are used to create Replicate and Join nodes, respectively.

- **SUBMODEL** is used to create a node that represents a previously defined submodel (atomic or composed).

- The remaining selections are used to connect nodes just as in other graphical editors.

### Submodel node

Submodels are the building blocks for constructing larger models. They are models that have already been built in any of the atomic or composed model editors, and form the leaves of the tree structure of Replicate/Join composed models. To create a Submodel node, first select **SUBMODEL** from the ELEMENTS menu or click on the icon in the toolbar. Then click in the editor pane where the node needs to be placed.

The set of commands available for a submodel node are available in its context menu (opened by right-clicking on the node) and are EDIT, OPEN, HIDE LABEL (or SHOW LABEL), and DELETE.

**EDIT** opens the Specify Atomic Model dialog as shown in Figure 6.3. To choose which model to use for this node, click the “...” button, and a list of existing atomic and composed models will be shown as in Figure 6.4. The name (or label) of the submodel can be set in the Node Name edit box.
6.1. REPLICATE/JOIN

Figure 6.3: Submodel type dialog.

Figure 6.4: List of available submodel types.
CHAPTER 6. COMPOSITION FORMALISMS

If the type of the submodel has been defined, the OPEN command opens the appropriate model editor for the submodel node while the composed editor is still open. This feature can be used to study the different building blocks, i.e., submodels of the composed model, while the model is being built.

The HIDE LABEL (SHOW LABEL) command hides (or shows) the label of the submodel that is shown below the node. Finally, the DELETE command deletes the submodel node from the composed model.

Join node

As mentioned before, a Replicate/Join composed model is structurally a tree, and it should be built in a bottom-up fashion. That means that the child(ren) of a Join or Replicate node should be defined before the node itself. Note that this composed editor allows for the definition of only one Replicate/Join tree, corresponding to one top-level (root) node. If there are nodes that are not in any way connected to the rest of the nodes, Möbius will show the error dialog shown in Figure 6.5.

A Join node establishes a number of state variable sharings among its children. It must have at least two children. Each sharing consists of a non-empty set of state variables of children of the Join node; no more than one state variable from each child is included. All the state variables in that set are superimposed and given a common name in the Join node. The common name is a state variable in the Join node.

To create a Join node, choose ELEMENTS→JOIN or click on the icon in the toolbar and then click in the editor pane where the node needs to be placed. You can access the set of commands available for a Join node by right-clicking on the node: the commands are EDIT, HIDE LABEL (or SHOW LABEL), and DELETE. HIDE LABEL, SHOW LABEL, and DELETE behave just as they did for submodel nodes.

After creating a Join node, you must connect it to each of its children by choosing one of the connection lines, STRAIGHT CONNECTION, CONNECTED LINE, or SPLINE CURVE from the ELEMENTS menu or the toolbar. A child of a
6.1. REPLICATE/JOIN

Join node can be another Join, a Replicate, or a submodel node. Note that the first point of the connection between a Join and its child must be the Join node. This direction determines which node is the child and which node is the parent.

In the next step, use the context-menu command EDIT to define the sharings established by the Join node. When you choose EDIT, the Define Join Node Dialog shown in Figure 6.6 appears. On the top of the dialog, the name of the Join node is shown. It can be modified in the Join node name edit box. There are also two lists in this dialog, called Join State Variables and Submodel Variables. The former, as its name implies, shows the state variables of the Join. Each line corresponds to one state variable, which itself corresponds to one sharing. If you highlight a Join state variable in the former list, the latter list shows the set of state variables of the children nodes (i.e., submodels) that are shared to create the Join state variable. For example, in Figure 6.6 the Join state variable computer\_failed is created by sharing 4 state variables among children of the Join node Computer. Notice the arrow symbol that is used to refer to a state variable of a submodel.

To create or edit an existing Join state variable, click on Create New Shared Variable or Edit Selected Variable, respectively. The Define Shared State Variable dialog, shown in Figure 6.7, will appear after you press either of the buttons.

The main part of the dialog is a multi-tab panel consisting of one tab for each

![Figure 6.6: Join node definition dialog.](image)
CHAPTER 6. COMPOSITION FORMALISMS

Figure 6.7: Join state variable definition dialog.

child of the Join node. To define a sharing, select the state variable to be shared in each of the child tabs. No more than one state variable can be selected from each tab. Finally, give the Join state variable a name by entering it in the Join node name edit box. This name will appear in the Join State Variables list (the left list) shown in Figure 6.6.

Three conditions must be satisfied before a set of state variables can be shared. First, they must all have the same initial value assigned to them in their corresponding model editors. If this condition is not satisfied, Möbius will show the error dialog shown in Figure 6.8(a). Second, they must all be of the same type, regardless of whether it is a simple type, like int, short, or char, or a structural type. An error dialog as shown in Figure 6.8(b) will appear if this condition is not met. Third, none of the state variables can be part of a currently defined sharing relationship.

The Share All Similar Variables button joins all state variables in the children of the Join node with common names. The Join state variable will have the same common name. If you choose the names of submodels’ state variables appropriately with respect to the larger composed model, this feature proves very useful.

Delete Selected Variable(s) deletes the Join state variable selected in the left
6.1. **REPLICATE/JOIN**

- Nonequal initial values error dialog.
- Non-matching types error dialog.

![Error dialogs related to the definition of Join nodes.](image)

Figure 6.8: Error dialogs related to the definition of Join nodes.

list. **Delete All Shared Variables** deletes all the Join state variables. The **Sort additions to list** checkbox controls how newly created state variables are added to the left list. If it is checked, the name of a new Join state variable will be added to the list in alphabetical order.

**Replicate node**

This node is a special case of the Join node. Its advantage over the Join node is that it leads to structural symmetries that can eventually result in less time- and space-consuming state-based numerical solution of the model. The extra restrictions that Replicate has relative to Join are as follows:

- All children of a Replicate are of the same submodel type. Therefore, in the composed editor, a Replicate node is connected to only one child node, and the number of replications is specified in the Replicate node definition dialog.

- Only state variables with the same name can be shared among replicas (copies) of a Replicate node. Therefore, to create a Replicate state variable, select only one state variable from the Replicate child; copies of that state variable will be shared among all replicas.

When you choose the **Edit** command from the context menu of a Replicate node, the **Define Rep Node** dialog, as shown in Figure 6.9, is shown. The label (name) of the Replicate node can be modified in the **Rep Node Name** edit box. The number of replications can be set in the **Number of Reps** edit box. That number can be an integral constant or global variable that has been defined in the current composed model or in any of the constituent atomic or composed models. Two lists of state variables (called **Unshared State Variables** and **Shared State Variables**) and a number of buttons between them are used to move state variables from one list to the other.
Each line in the latter (right) list shows the name of a state variable that has been shared among all replicas of the Replicate node. The rest of the state variables of the child submodel are shown in the left list. As mentioned before, each Replicate (or Join) node is itself a model. Notice that in the corresponding model of the Replicate node, for each name in the right list, there is only one copy of that state variable, and for each name in the left list, there are as many copies as have been specified in the Number of Reps edit box, i.e., one copy per child replica.

The buttons between the lists are self-descriptive. Share moves the selected state variable in the left list to the right list and Unshare does the reverse. Share All moves all state variables from the left list to the right list, and Unshare All does the reverse.

6.2 Graph

This section covers the overview of the Graph composition formalism supported in Möbius. It includes a brief description of the different parts of the Graph composed model editor and basic steps to build a composed model.

6.2.1 Overview

The Möbius tool also supports another composed model formalism called Graph composition [26,35]. While the structure of a Replicate/Join composed model is
a acyclic tree, the structure of a Graph composed model has no restrictions. Here, a Join node linking two models indicates an equivalence-sharing relationship between the two models, as in a Replicate/Join composed model. Much like Replicate/Join composition, lumping techniques based on computational group theory can be used to find all symmetries in the graph structure of the model automatically [26]. Exploiting these symmetries leads to efficiencies in the solution process. However, the graph theory used to detect these symmetries is currently not implemented in Möbius. Thus the Graph composition formalism, as it stands in Möbius, provides no advantage over the Replicate/Join composition formalism.

6.2.2 Graph composed model editor

Figure 6.10(a) shows the Graph composed model editor. As with other model editors, this editor can be opened either by creating a new model or by opening an existing one. To create a new Graph composed model, select Composed or any of its children in the project window and select the NEW command either by choosing it in the context menu or by clicking the leftmost icon in the toolbar (see Figure 3.1). To open an existing Graph model, select the OPEN command either by choosing it in the context menu or by clicking the corresponding icon. Because the

![Graph composed model editor](image1)

(a) Graph composed model editor.

![The same model represented in the Rep/Join formalism.](image2)

(b) The same model represented in the Rep/Join formalism.

Figure 6.10: A composed model represented in two different composition formalisms.

Graph composed model editor is identical to the Replicate/Join composed model editor without a Replicate node, please refer to Section 6.1.2 for a discussion of
this editor. As shown in Figure 6.10, any model expressed in the Graph composed model formalism can also be expressed in the Rep/Join formalism.

### 6.3 Action Synchronization

This section covers the overview of the Action Synchronization formalism supported in Möbius. It includes a brief description of the different parts of the Action Synchronization model editor and basic steps to build a composed model with this formalism.

#### 6.3.1 Overview

The Möbius tool also supports the composed model formalism known as Action Synchronization. Much like the Replicate/Join composed model formalism, Action Synchronization takes the form of an acyclic tree, utilizing join nodes in a similar manner as Replicate/Join.

Action Synchronization is a composition formalism based on the notion of composing actions instead of places. If two actions are said to be shared between two models, then the new composed action’s enabling conditions are the union of the enabling conditions of the submodel actions, that is neither action can fire unless all are enabled. When the composed action fires, the result is then the union of results of the submodel actions.

#### 6.3.2 Action Synchronization model editor

Figure 6.11(a) shows the Action Synchronization composed model editor. The editor is accessed either by creating a new model, or opening an existing model, much as with the other composed editors.

Due to the similarities of Action Synchronization to the Replicate/Join composed model editor, please refer to Section 6.1.2 for a discussion of most of the features of this editor. The primary difference comes when editing a join node itself.

Unlike the Replicate/Join composed model editor, Action Synchronization connects models by Joining on actions, not places. As such when the Jode node editor (Figure 6.11(b)) is opened, instead of an option to “Create New Shared place”, we have “Create New Shared action”.

The shared action dialog, seen in Figure 6.12, allows one to define synchronization amongst actions. Unlike sharing variables, shared actions need not have the same rate. The new shared action instead has the rate defined by the user in the
6.3. ACTION SYNCHRONIZATION

(a) Action Synchronization Model Editor.

(b) Action Synchronization join node editor.

Figure 6.11: Parts of the Action Synchronization Model Editor.

Figure 6.12: Shared action dialog.
“Rate” dialog. The actions to be shared in each of the submodels are then selected from the tabbed section of the dialog.

If the actions to be shared all have common names, the **Share All Similar Action** button can be used. If this is done the Join action will have the same common name. After sharing all similar actions, the shared actions must be edited with the **Edit Selected action** button to set the rate to an appropriate value.

Much as with the Replicate/Join editor, the button labeled **Delete Selected Actions** deletes the selected Join action from the list on the left, and **Delete All Shared action** deletes all of the Join actions.
Chapter 7

Reward Formalisms

Reward formalisms define functions that measure information about the system being modeled. Currently, Möbius provides one reward formalism: performance variables.

7.1 Performance Variables

The Performance Variable Editor is shown in Figure 7.1. It consists of a menu bar and a split pane. The left side of the split pane contains a tabbed panel with two tabs: Performance Variables and Model. The right side of the split pane contains the editable fields for the currently selected performance variable.

The Performance Variables tab is the main display and is used to create new variables or select an existing variable so it can be edited, renamed, copied, or deleted.

The Model tab lists the top-level model on which this reward model is built. Often, the top-level model is a composed model, in which case the available submodels within the composed model are listed in the lower table. The top-level model is also referred to as child model of this reward model.

In addition to the menu options that are common to all Möbius editors (see Section 4.1), the following operations are available within the performance variable editor, via the main menu, buttons at the bottom of the left panel, or a right-click pop-up in the variable list:

- **Add Variable:** Type the name of the new variable and click the Add Variable button (or hit the < Enter > key). The text in the new variable name text field disappears automatically when you type the new variable name.
CHAPTER 7. REWARD FORMALISMS

Figure 7.1: Performance Variable Editor.

- **RENAME**: Change the name of the selected variable.
- **MOVE UP**: Move a variable up in the variable list. Allows for the grouping of related variables.
- **MOVE DOWN**: Move a variable down in the variable list. Allows for the grouping of related variables.
- **COPY**: Copy the selected variable to a new variable with a new name.
- **DELETE**: Delete the selected variable.

### 7.1.1 Variable definition

When a variable is selected in the variable list, the right side of the editor displays the variable definition panel. At the top of this panel is the name of the variable currently selected. Beneath the name is a tabbed pane used to define specifics of
7.1. PERFORMANCE VARIABLES

the variable. The tabbed pane contains five tabs: Submodels, Rate Rewards, Impulse Rewards, Time, and Simulation.

7.1.2 Submodels

The Submodels tab lists the names of all of the models in the child of the reward model. You must apply each reward variable to one or more of the available submodels. You can do so by selecting the model in the list. The <Ctrl> key can be used to select multiple individual items. The <Shift> key can be used to select a range of items.

When Möbius computes the reward function, it will do so on each instance of the selected models. For example, if there are $N$ instances of the selected submodel in the top-level model, the reward function will be evaluated $N$ times. For some types of reward functions, it is desirable to divide the reward function by the number of instances in the model ($N$ in this example), so that the reward represents the average of the $N$ instances.

With the current version of Möbius, you must use caution when defining rewards that depend on the state of multiple submodels. When multiple instances of the specified submodels are created by replication in the composed model, the results obtained are often nonintuitive. For example, consider a child model that is a composed model with 2 instances of submodel “A” and 3 instances of submodel “B”. If a reward is defined on state variables from both A and B, one might expect that the function would be evaluated five times (once on each A, and once on each B). However, that is not the case. Instead, the function will be evaluated a total of 6 times, once for each possible pair of an A with a B. (This behavior might change in future versions of Möbius.)

Selecting the submodel populates the Available State Variables panel in the Rate Rewards tab with the names of all state variables found in the selected submodel(s). Similarly, it also populates the Available Actions panel in the Impulse Rewards tab with all of the actions found in the selected submodel(s).

7.1.3 Rate rewards

The Rate Rewards tab is used to define rewards based on the time in each state. The top panel lists the Available State Variables, based on the submodels selected in the Submodels tab. The bottom panel contains the Reward Function. The reward function defines the measurement this rate reward should take. The reward function is written as a piece of C++ code, and should end with a return statement returning the value of the function.
As a shortcut for the user, double-clicking in the top panel inserts the state variable name at the location of the text cursor in the bottom panel. The value of the reward must be returned using the state variable value access function. This function is formalism-specific. For example, the value access function is $\text{Mark}()$ for places in SANs and buckets in Buckets and Balls. Refer to the appropriate formalism documentation for details on access functions and other available functions that could be used to define rewards.

7.1.4 Impulse rewards

The Impulse Rewards tab defines reward functions that are evaluated when actions in the child model fire. The top panel contains a table showing the name of each action in the child model, and a column specifying whether or not an impulse function is defined for the action. To define an impulse function, click on the name of the function, and then write the C++ function for the impulse reward in the lower panel. The code should return the function result using the C++ return statement.
7.1. PERFORMANCE VARIABLES

7.1.5 Time

In order to solve for the reward measures via either simulation or numerical solution techniques, Möbius requires the specification of additional parameters for each reward variable. These parameters define the type of results to measure for the specific times of interest.

Reward variables can be defined as one of several different types. The type of the reward variable determines when, in system time, the reward function is evaluated. Evaluation times can be specified manually using a table, as shown in Figure 7.3, or as an incremental range, as shown in Figure 7.4. The possible reward variable types are:

- **Instant of Time**: The reward function is evaluated at the specified point in time. The desired time is specified in the **Start time** field. Units of time are the same units used for the parameters for the action distributions in the atomic models.
CHAPTER 7. REWARD FORMALISMS

Figure 7.4: Definition of time points using incremental range options.

- **Interval of Time**: The variable returns the weighted sum of all of the values of the reward function, where each value is weighted by the amount of time the value is in existence, between the starting and ending times of the specified interval. The desired start and stop times for the interval are specified in the **Start** and **Stop** text fields.

- **Time Averaged Interval of Time**: The variable returns the interval of time result, divided by the length of time for the interval. As with interval of time variables, the desired start and stop times for the interval are specified in the **Start** and **Stop** text fields.

- **Steady State**: The reward function is evaluated after the system being modeled reaches steady state. The steady state simulation algorithm used is referred to in literature as *batch means* (see [1]). This approach assumes that there is an initial transient period that must pass before the system reaches its steady state behavior. Once the system is in steady state, the algorithm evaluates the reward function multiple times to gather the observations to compute the statistics. This technique is appropriate when enough time occurs between the samples to permit the assumption that the samples are independent of each other.

Simulation using batch means is typically more efficient than standard sim-
7.1. PERFORMANCE VARIABLES

Figure 7.5: Impulse reward definition.

ulation approaches, since the possibly long initial transient period must be processed only once in batch means for all the observations, while traditional simulation would require processing of the initial transient period for each observation.

7.1.6 Simulation

The Simulation tab is used to define two aspects of reward variables that are unique to simulation. They are variable estimation and confidence interval definition.

Estimation

When a model is being solved via simulation, the system is executed multiple times using different randomly generated event streams. Each execution generates a different trajectory through the possible event space of the system, due to the differences in the order and choice of events that occur. The reward variables are
evaluated for each trajectory to create an observation. Statistical estimates of the reward variable value are then computed from the observations.

Multiple estimates can be computed for each variable. Möbius supports four basic estimations: mean, variance, the probability that the function is in an interval, and the probability distribution (and density) functions. To enable the computation of any of these types, click the appropriate check box. Additional parameters are required for intervals and distributions.

When Estimate Interval is selected, the four interface components beneath it are enabled. The Lower Bound and Upper Bound fields are used to specify the lower and upper bounds of the interval. The Include Upper (Lower) Bound checkboxes determine whether the upper (lower) bound itself is part of the interval.

When Estimate Distribution is selected, the four interface components beneath it are enabled. The Lower Bound and Upper Bound text fields specify the lower and upper limits of the distribution that will be measured. The Step Size determines the width of each bin in the discrete representation of the distribution. The number of samples in the distribution is computed by \((Upper - Lower)/\text{StepSize}\). Varying those three parameters makes it possible to focus distributions on specific areas of the reward function space, with varied resolutions.

If Estimate out of range probabilities is selected, the probability that the reward function value will be lower than the lower bound of the distribution will be computed, as will the probability that the reward will be above the upper bound of the distribution.

**Confidence**

In order to get statistically significant estimations of the reward variables, it is necessary to generate many trajectories. In order to give an estimate of the accuracy of the calculated estimates, confidence intervals are computed as the observations are collected. When the simulation reaches the desired confidence level for every variable, the simulation will stop. (The simulator will also stop if it reaches a maximum observation limit without achieving the desired confidence level.)

Three parameters define the confidence interval. The Confidence Level text box specifies the desired probability that the exact value of the reward variable will be within the specified interval around the variable estimate. The Confidence Interval text box specifies the width of the acceptable interval around the variable estimate. The interval can either be Relative to the variable estimate, or be an Absolute number. For instance, a relative confidence interval of .1 and a confidence level of .95 for a mean variable will not be satisfied until the confidence interval is within 10% of the mean estimate 95% of the time.
Chapter 8

Study Editors

Often a modeler wishes to investigate the behavior of systems for several different parameter values, perhaps corresponding to different system configurations. Möbius provides a convenient method to do so through the creation of studies. Recall from Chapters 5 and 6 that global variables can be defined on atomic or composed models. A study allows one to examine the effect of varying parameters (global variables) on system performance. Within a study, one or more experiments may be defined based on the different values the parameters may take. More precisely, an experiment is one tuple of parameter values for which a model may be solved.

Möbius currently supports two types of studies: range studies and set studies. Keep in mind that even if no global variables have been defined in a model, a default study must still be created. A new study can be created by right-clicking on the Study node from the Möbius project tree view. The study editor selection dialog allows users to choose the appropriate types of study editor.

8.1 Range Study

A range study allows each global variable to be assigned either a fixed value or a range of values. In a range study, the experiments are created automatically by Möbius as the cross product of all possible values the global variables may take. For example, if there are two global variables, each of which is assigned a range of six values, the study will be composed of 36 experiments. The editor for a newly created range study is shown in Figure 8.1. The name of each global variable defined in the child model is displayed in the table, along with its type and value (initially zero). You can reorder or resize the columns by clicking and dragging on the appropriate column header or border.
To enter a fixed value for a variable, simply click in the corresponding Variable Value text box and enter the value. To enter a range for a variable, click on the appropriate button at the bottom of the dialog (Incremental Range, Functional Range, Manual Range, or Random Range). An example of a completed range study can be found in Figure 8.2. Note that if a variable is assigned a fixed value, that value is displayed in the table. Likewise, if a variable is assigned a range of values, the type of range is displayed.

8.1.1 Incremental range

An incremental range is specified with a starting value, an ending value, and an increment, which may be additive, multiplicative, or exponential. Figure 8.3 shows the dialog. The first three elements of the dialog show the name of the study, the name of the variable being assigned, and the type of this variable.

A variable defined with an incremental range will take on all values between Initial and Final at increments of Increment. That is, if $a$ represents the initial value, $b$ represents the final value, and $i$ represents the increment, the variable will take on values for the different increment types according to Table 8.1. Click on
8.1. RANGE STUDY

8.1.1 Incremental Range

Figure 8.3: Incremental Range dialog.

the View Values button to see a list of all the values the variable will take on in this range. The list may be exported to a text file.

Table 8.1: Three types of incremental ranges.

<table>
<thead>
<tr>
<th>Increment Type</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Additive</td>
<td>( a, a + i, a + 2i, \ldots, b )</td>
</tr>
<tr>
<td>Multiplicative</td>
<td>( a, a \cdot i, a \cdot i^2, \ldots, b )</td>
</tr>
<tr>
<td>Exponential</td>
<td>( a, a^i, a^{i^2}, \ldots, b )</td>
</tr>
</tbody>
</table>

8.1.2 Functional range

A functional range is essentially a function of an incremental range. An incremental range is defined as in Section 8.1.1 to be the domain, and the functional range is obtained by applying a function to every element in the domain. Thus, the top half of the Functional Range dialog is identical to the Incremental Range dialog. The function may be a standard function, such as \( e^x \) or \( \sin x \) using the Range f(x) option, or a specialized function (e.g., quadratic, exponential, or linear) using the Range f(x,n) boxes.

For example, if the incremental range shown in Figure 8.3 was used as the domain and \( f(x) = x^2 \) was used as the function, the result would be the dialog in Figure 8.4 and the values in Table 8.2.
Figure 8.4: Functional Range dialog.

Table 8.2: Functional range example.

<table>
<thead>
<tr>
<th>Domain $x$</th>
<th>Range $f(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.0</td>
<td>25.0</td>
</tr>
<tr>
<td>10.0</td>
<td>100.0</td>
</tr>
<tr>
<td>15.0</td>
<td>225.0</td>
</tr>
<tr>
<td>20.0</td>
<td>400.0</td>
</tr>
<tr>
<td>25.0</td>
<td>625.0</td>
</tr>
<tr>
<td>30.0</td>
<td>900.0</td>
</tr>
</tbody>
</table>
8.1.3 Manual range

Specify a manual range by entering each value in the range separately. To add a value to the range, enter the value in the New Value text box and click Enter. The values in the range are displayed in the Current Values box. The five buttons to the right of this box can be used to manipulate the values in the range as follows:

- **Up**: Move a value up in the list
- **Down**: Move a value down in the list
- **Delete**: Delete the selected value from the list
- **Delete All**: Delete all values in the list
- **Import**: Import values from a text file

The incremental range in Figure 8.3 could equivalently be represented by the manual range in Figure 8.5. For this example, it would be more convenient to use an incremental range than to enter each value in the range manually. However, a manual range is necessary if you wish to specify range values that do not follow a regular pattern.

8.1.4 Random range

A random range allows you to assign random values to global variables. Enter into the Number of Samples text box the number of random values to generate within
the range defined by Minimum Value and Maximum Value. Figure 8.6 shows an example in which the variable \textit{arr\_rate} is assigned 6 random values between 5 and 30.

Figure 8.6: Random Range dialog.

Clicking the View Values button will display the pseudorandomly generated values, which may be exported to a text file. To create new values using a different seed, click the Regenerate button. The resulting values will be the actual values for the global variable used by the solver.

8.2 Set Study

A \textit{set study} allows you to enter vectors of fixed values for each global variable. Thus, unlike a range study, in which experiments are created automatically, a set study has experiments that are defined explicitly by the user. Set studies are useful when the global variables do not change in a regular pattern with respect to each other. The editor for a newly created set study appears in Figure 8.7. All the global variables from the child model and their types are displayed in the table on the left. The Experiments table on the right shows the names of all the experiments that have been created as well as the values for the global variables in each experiment. Möbius requires that experiment names be distinct.

When you create a new set study, one experiment is initially defined in which all global variables default to zero, as shown in Figure 8.7. To change the values of the global variables in an experiment, click inside the appropriate box in the column corresponding to the experiment and type the value. The buttons at the bottom of the dialog give you the ability to manage experiments, and are described in the following paragraphs.
8.2. SET STUDY

Fig 8.7: New Set Study Editor.

Add  Click the Add button to add an experiment. Type the name of the new experiment at the prompt. Although the default name for the first experiment is “Experiment 1,” it can be renamed as explained below. In general, it may be helpful to give experiments more descriptive names.

Delete  To delete an experiment, click anywhere in the corresponding column of the Experiments table, and then click the Delete button. A prompt will ask you to confirm the deletion. Note that an experiment can be deleted only if multiple experiments exist in the study. If an experiment is the only one in the study, it cannot be deleted, since every study must contain at least one experiment.

Copy  To create a new experiment that has the same values for the global variables as an existing experiment, click anywhere inside the column for the original experiment and then click the Copy button. You will then be prompted for the name of the new (duplicated) experiment. While there would be no reason to have two identical experiments, using the Copy button is convenient when creating an experiment that is identical to another except for a few values, especially when the number of global variables is large.

Rename  Click the Rename button to rename an existing experiment. Type the new name of the experiment at the prompt and hit OK.

Import/Export  The Import and Export buttons are used, respectively, to import an experiment from a text file and export an experiment to a text file. The latter option requires that you select an experiment for export. The “import” function reads in files in the same format into which “export” writes them. Thus experiments...
may be exported for later import into a (possibly different) study in either Möbius or UltraSAN.

8.3 Experiment Activator

As mentioned previously, a study consists of one or more experiments. Both study editors include the ability to activate or deactivate experiments via the Experiment Activator button (see Figure 8.8). To the left of this button in the dialog, the number of active experiments is displayed. An active experiment is one set of parameters for which the model will be solved when solution is performed (see Part III). In contrast, an inactive experiment will be skipped during solution. Use this function when only a subset of the experiments are to be carried out.

Figure 8.8: Completed set study example.

Figure 8.9 shows the dialog that will appear if you click the Experiment Activator button. At the top can be found the name of the study, the total number of experiments, and the number of them that are active. The table below lists the global variables and their values in each of the experiments. The check boxes underneath the experiment names indicate whether the experiment is active or inactive. All experiments are initially marked as active. To deactivate an individual experiment, click in the check box underneath its name. Click the Activate All or Deactivate All button to activate or deactivate all experiments at once. Then hit OK to commit or Cancel to discard changes.
8.3. EXPERIMENT ACTIVATOR

Figure 8.9: Experiment Activator dialog.
Chapter 9

DOE Study Editor Interface

9.1 User Interface Flow

Figure 9.1: Flow of model solution with the DOE study editor.

Chapter 2 presented the modeling flow in Möbius. The first steps in the modeling process are the definitions of the atomic, composed, and reward models. It is
at these three modeling levels that all global variables are defined, but not assigned values. The study level is where global variables are assigned values. Figure 9.1 shows the modeling flow from the study level onward for a DOE study. During the study definition, all global variables are assigned values for each experiment. After a study is defined, it is saved and compiled into object code, which is later linked in by the solver during the model solution step. After all experiments are complete, the DOE study is reopened, and the results from the experiments are analyzed. If necessary, the study can be modified, and new experiments can be performed. This iterative process continues until the desired results are achieved. The remainder of this chapter will present the details of the graphical user interface of the DOE study editor.

9.2 Defining a DOE Study

Figure 9.1 shows that the first step of the DOE experimentation process is to define a DOE study. There are several steps in the study definition process. Figure 9.2 presents the flow of events involved in defining a DOE study. The remainder of this section will present the details of each step of the DOE study definition.

![Diagram of DOE study definition process]

Figure 9.2: The flow of events in the DOE study definition process.

The creation of a new DOE study is similar to the creation of any new model in Möbius. First, from the Möbius project window, select the type of module. Select the study node by clicking on the tree node named “Study” and then press the New button. A dialog asking for the study name and type is presented. Figure 9.3 shows a dialog from the creation of a study named “Workstation” of type “Design
of Experiments Study.” After pushing the OK button, select the reward model upon which the study will be based. After selecting the appropriate reward model and pushing OK, you will be presented with the Constant Global Variable Selection dialog.

The purpose of the Constant Global Variable Selection dialog is to help users select which global variables are to be held at constant values throughout the study. There are several reasons why some global variables would remain fixed to specific values during a study. One reason is that the variation of some model parameters may be relevant to another study, but not to the current one. Another reason to fix global variables to constant values is that a previous DOE study may have revealed that the variables do not significantly affect the reward model under study, so can be fixed to convenient values. Alternatively, the optimal values of some global variables may already have been determined from earlier studies. In that case, the global variables can be fixed to their optimal values. Any global variables that are held constant will not be included in the empirical model of the response created by the DOE analysis.

Figure 9.4 shows a Constant Global Variable Selection dialog with a list of all global variable names and types defined in the underlying model. To designate a global variable as constant, its checkbox in the Constant column must be selected by a single mouse click. Once that box has been selected, the value can be entered into the corresponding row under the Constant Value column. There are two ways to enter values into a table in the DOE study editor. One is to single-click the table cell and simply start typing the value. The other is to double-click the table cell. A blinking caret will appear, and all text will be left-justified while you are typing.
Regardless of which method is used, pushing the \(< Enter >\) key will commit the new values to the table. After being entered, all numbers will be right-justified in their table cells. When all of the constant global variables have been selected, push the OK button to proceed to the next step. At any point during the study definition process, you may push the Cancel button in order to cancel the study definition process. After pushing the Cancel button, you will be asked whether the study is to be saved. If it is, it can be reopened later, and the definition process can resume. If you choose not to save the study, then it will be permanently discarded.

![Constant Global Variable Selection Dialog](image)

**Figure 9.4:** Constant Global Variable Selection Dialog.

The next step in the DOE study definition is the selection of the design type. Figure 9.5 shows the Design Type Selection dialog. The two categories of designs, factorial and response surface, are presented in separate tab panels. Selecting the Factorial tab shows a list from which either a Plackett-Burman or a two-level factorial design may be selected. Selecting the Response Surface tab displays a list with a choice of either a Central Composite or a Box-Behnken design. After selecting the desired design type from the appropriate list, you may either proceed to the next step by pushing the OK button, or return to the previous dialog by pushing the Back button. At any point in the study definition process, you may move forward or backward to any step and make changes to the study.

After you have selected the design type, a dialog will prompt you to enter parameters for the chosen design. All design types require that both lower and upper bounds be specified for each nonconstant global variable. In factorial designs, global variables are varied over exactly two values. The values are equal to the low and high values entered in the design parameters dialog. Figure 9.6 shows an example 2-Level Factorial Design Parameters dialog. The top table lists each nonconstant global variable name and type. A low and high bound must be entered...
9.2. DEFINING A DOE STUDY

Figure 9.5: Design Type Selection dialog.

Figure 9.6: Design parameters dialog.
for each variable. In two-level factorial and central composite designs, a second table displays a list of the different designs that are available. You should select the design that best suits the available computing resources. A single mouse click will highlight the row of the selected design. In Figure 9.6, the full resolution design is selected. After all global variable bounds have been entered and the design fraction is selected, the study definition may proceed to the response selection step.

![Response selection dialog](image)

Figure 9.7: Response selection dialog.

A reward model may consist of several reward variables. Each of the reward variables can be solved using a variety of solution methods available in Möbius. For example, the mean and variance of a reward variable measuring workstation performance may be calculated by a simulation or via several numerical solution methods. The solution from each solver can be represented by a unique response. Each response can be analyzed separately, allowing for the comparison of different solution methods on the same reward variable.

Figure 9.7 shows a Response Selection dialog. A checkbox near the top, labeled Edit responses separately for each reward variable, allows a different set of solution methods to be used for each reward variable. Usually, this checkbox is not selected, and all reward variables are solved using the same solution methods. A scrolling panel in the middle of the dialog lists all of the solver types available in Möbius. The user selects the desired solution methods by clicking the checkbox.
9.3. **DOE STUDY EDITOR WINDOW**

9.3 DOE Study Editor Window

After a study is defined, the **DOE Study Editor** window appears. It is the main window and is always present, except during the study definition process. Figure 9.8 shows an example **DOE Study Editor** window. The data is from the fictitious workstation model from Chapters 2 and 3. The window has four areas. The first is the menu bar at the top. The menu functionality will be discussed later in the chapter. At the bottom is the logo panel, which is similar to the logo panel found in other Möbius modules. The lower central portion of the logo panel contains a status area that occasionally displays messages in red text about the status of an activity.

On the left side of the window is the design tree panel. The selected node of the design tree determines what is displayed in the main panel, which is the area directly to the right of the design tree. The design tree panel and main panel are separated by a movable divider. The mouse can be used to drag the divider left and right in order to give more screen area to a particular panel. At the top of the divider, as shown in Figure 9.8, there are two small triangles. Clicking one of the triangles moves the divider all the way to the edge of the window, causing either...
the design tree panel or the main panel to be hidden, depending on which triangle was clicked. If a panel is hidden, it can be revealed by a single click anywhere on the divider.

The graphical user interface for the DOE study editor was written using Java Swing classes. The appearance of the GUI varies depending on the operating system being used. The screen shots in this chapter were taken on the Microsoft Windows operating system. The DOE study editor will appear slightly different on other operating systems, such as UNIX or Linux. However, the functionality described in this manual will be consistent across operating systems.

9.4 Design Tree

Most of the time spent using the DOE study editor involves analyzing the contents of the main panel. There are five different views available for display in the main panel. Each of the five views corresponds to one of the five different types of nodes in the design tree. Clicking on an unselected design tree node will change the contents of the main panel. This section gives a brief description of each design tree node.

At the top of the design tree is the Notes node. Selecting the Notes node displays an editable text area in the main panel. Any comments entered in the text area will be saved when the study is saved. The text area supports cut, copy, paste, undo, and redo functionality, either from the Edit menu or by keyboard shortcuts. Cut, copy, paste, undo, and redo are performed by hitting <Ctrl>-<x>, <Ctrl>-<c>, <Ctrl>-<v>, <Ctrl>-<z>, and <Ctrl>-<Shift>-<z>, respectively.

Below the Notes node is the Design node, which, when selected, displays the design panel in the main panel. The design panel presents all input data for the study, including all global variable and response values for each experiment. The design panel will be discussed in Section 4.6.

Selecting the Summary node displays a design summary, an example of which is shown in Figure 9.9. Response and global variable data are summarized in HTML-formatted tables. If the design is a two-level factorial type with less than full resolution, the alias structure is presented at the bottom of the summary panel. The text from the summary panel can be exported to a file using the Design menu’s EXPORT TEXT menu item. Further details of all menu items will be offered in the next section.

The Analysis node contains a brief summary of the analysis procedure. Each response in the design panel has a corresponding tree node under the Analysis node. Response analysis requires several steps. Most of the functionality of the DOE study editor is used during response analysis. Before any analysis can be
9.4. **DESIGN TREE**

Figure 9.9: Summary panel.
performed on a particular response, the user must enter response values for all experiments into the design panel. Details on how to do so will be presented in Section 4.6.

9.5 Menus

There are four menus in the DOE study editor. The first is the File menu, which performs the same functions as the File menus in other Möbius editors. The SAVE item saves any changes to the study to disk and compiles the global variable values for each experiment into object code. A dialog appears that shows the progress of the compilation and linking process. The object code will be linked in by the solver during model solution. The DOCUMENT menu item saves an HTML description of the study in the study directory. The CLOSE menu item closes the DOE study. If any changes were made, you will be prompted to decide whether the changes should be saved.

The Edit menu provides basic cut, copy, and paste functionality in any editable text area or text field. Undo and redo functionality are also available, but only in the notes panel, as discussed earlier. Keyboard shortcuts are available for all menu items.

The Design menu contains items specific to the DOE study editor. The EXPORT REGRESSION MODEL menu item is only enabled when an appropriate analysis panel is displayed. The uses of exporting the regression model will be discussed in Section 9.8.2.

The EXPORT TEXT menu item is used to save any displayed panel to a text or HTML-formatted file. If the notes or design panel are displayed, the contents of the panel will be saved to a text file. If the summary panel is displayed, the contents of the panel will be saved to an HTML file. Other uses of the EXPORT TEXT menu item will be presented later.

The REDEFINE DESIGN menu item is used to initiate the study definition process explained in Section 9.2. A user might wish to redefine a study to correct a mistake made during the initial study definition. He or she could also use redefinition to augment the design. Design augmentation involves adding more experiments to a study so that more information can be gained about the responses under study. For example, a half-fraction two-level factorial design may not present enough information for the user to build an accurate empirical model of the response. The design can be augmented with the other half-fraction in order to create the full-resolution design, doubling the number of experiments in the study. However, half of the experiments will already have been completed. After the remaining half are run, an analysis of the full resolution 2k design can be performed.
One caveat of design redefinition is that all data in the response columns in the design panel are erased when a study is redefined. In order to restore the data, the user must re-query the results database to fill in the response columns. Instructions for gathering response data from the results database will be presented in the next section.

![CentralCompositePreferences](image)

Figure 9.10: Preferences dialog.

The HIDE/SHOW LOGO PANEL menu item is useful for gaining valuable screen area. On small screens, it may be difficult to view some of the graphs in the analysis panels. Hiding the logo panel can increase the area available for viewing such graphs. That can reduce the amount of scrolling necessary in viewing the data. One disadvantage of hiding the logo panel is that the status bar will not be visible; furthermore, any important messages displayed there would also not be visible.

The last item in the Design menu displays a list of preferences for the current study. Figure 9.10 shows an example Preferences dialog. The first two items allow the user to specify the preferred window dimensions. Once the preferred height and width have been entered and the study has been saved, the DOE Study Editor window will reopen with the specified window dimensions. This feature allows users to customize the window size to suit a particular large or small screen. The remaining preference items are specific to the analysis procedure and will be discussed later. It is important to note that saved preferences are relevant only to a specific study, not to all DOE studies.

The last menu is the Help menu on the right side of the menu bar. The ABOUT item displays information about copyrights and contact information for the PER-
FORM research group. The REWARD INFO item displays the name of the reward model on which the study is based.

### 9.6 Design Panel

Whenever a DOE study is opened, the first view displayed in the main panel is the design panel. The design panel displays the entire experimental design in the design table. Figure 9.11 shows an example design table. Each row represents an experiment. The global variable columns represent the factor values. The reward variable columns represent the responses. The Active column displays the experiments that will be performed by the solver, and replaces the Experiment Activator found in the Set and Range study editors. If a checkbox is selected, then the experiment for the corresponding row is active.

You can activate several popup menus by right-clicking on certain column headers of the design table. The Active column has a popup menu with four choices for experiment activation. All experiments can be either activated or deactivated with a single menu choice. Alternatively, experiments from only the selected rows can be activated or deactivated. Row selection is performed by left-clicking on a row with the mouse. The <Shift> and <Ctrl> keys can be used with the mouse click to select a sequence or noncontiguous block of rows, respectively.

For response surface designs, a popup menu is available on the Experiment column header. The user may add experiments to the bottom of the table. Any user-added experiments may be removed from the table via the REMOVE SELECTED item on the Experiment column’s popup menu. Only user-added experiments may be removed from the design table; the built-in design must remain intact. The ability to add experiments to factorial designs is not enabled. The factorial designs built into Möbius are orthogonal. Adding arbitrary experiments would destroy the orthogonality and distort the analysis. Response surface designs do not require
orthogonality for analysis. You should avoid creating arbitrary experiments that are far from the design space of the original built-in design. Points far from the original design center can grossly distort the response surface, resulting in inaccurate prediction. For the same orthogonality reason, global variable values cannot be edited in the design table for factorial designs. They can, however, be edited in response surface designs. Again, you are cautioned against entering factor levels that produce design points far from the original design.

You can activate the response popup menu by right-clicking on any reward variable column header. The Insert Response item allows you to add a single response column to the table. The dialog in Figure 9.12 appears, prompting you to select the reward variable, the solver type, and the reward type. Pushing the OK button in the example shown in Figure 9.12 would add a new response measuring the mean performance using the adaptive transient solver. The Delete Response item removes a response column from the table. There must be at least one response, so the last response cannot be deleted.

There are three menu items in the response popup menu associated with filling the column with data. One item, which is very useful for experimenting with the DOE study interface, is the Fill With Random item. Selecting that menu item will fill the response column with uniformly distributed integers between 90 and 110. No analysis of a response can take place until the entire response column is filled with data. Therefore, the Fill With Random feature is a quick way to start playing with the analysis tools, which will be described later. To analyze real data, you must fill the response columns with solver data by querying the results database for the model solutions.

There are two popup menu items for filling a response column with data from the results database. The Fill Selected item fills in data only for the selected rows in the design table. The Fill Column item fills the entire column with the solved reward variable results. Selecting either of the two choices causes a dialog...
similar to the one in Figure 9.13 to appear. Because the column associated with the popup menu represents data from a simulator, the dialog in Figure 9.13 will prompt you to select from a list of simulators defined in the project. Once the solver has been selected, the results database will be queried, and the response values will be retrieved and placed into the response column.

![Figure 9.13: Solver selection dialog for results database query.](image)

To erase values from a response column, the CLEAR SELECTED or CLEAR COLUMN popup menu items can be selected. The CLEAR SELECTED item clears the response values only for the selected rows in the table. The CLEAR COLUMN item erases the entire column. The user may also edit the response values by typing the values directly into the table cells. Only after every response value in a column has been entered may analysis of the response be performed.

### 9.7 Analysis Procedure

For every response column in the design table, there is a corresponding node under the Analysis node in the design tree. After all response values have been filled in for a particular response column, the corresponding tree node can be activated. Clicking on the response node reveals the analysis tabs in the main panel. There is a sequence of six tabs that, when clicked, reveal a panel for performing one step in the analysis procedure. The user interface for the tab panels will be presented in the remainder of the chapter. Figure 9.14 shows the analysis flow corresponding to the six tab panels. In summary, the goal is to build and evaluate a regression model of the response and to use the regression model to predict response values at points in the design space other than the experimental points. Steps 1 and 2 transform the observed response values (if necessary) and build a regression model from the significant effects. Steps 3 and 4 analyze the model for significance and test the assumptions made on the data. If the model is good, then it can be used to produce graphs of the response and predict new values in the design space, as indicated in steps 5 and 6 of Figure 9.14. The six tabs are shown along the top of Figure 9.15. They progress from left to right as the analysis progresses.
Figure 9.14: Analysis flow in the DOE study editor.
CHAPTER 9. DOE STUDY EDITOR INTERFACE

Figure 9.15: Six tab panels in analysis procedure with Transformation tab selected.
9.7. ANALYSIS PROCEDURE

The first step in the analysis procedure allows the user to transform the observed response values. After a transformation is applied, all analysis is done on data that are some function of the original solver values. There are three situations in which a transformation may be required. The first occurs when the maximum-to-minimum ratio of the response values is large. The second occurs when a priori knowledge of the system results in the belief that a particular function of the response, rather than the response itself, is better suited to a regression model. Both of those situations require that the transformations take place before any analysis is performed. It is for that reason that the Transformation tab is placed first in the analysis flow.

The third situation in which a transformation is required occurs after an initial analysis has revealed that the assumptions on the response data have been violated. For example, a nonconstant variance might not be discovered until the Diagnostic Plots phase of the analysis. The user must then backtrack to the Transformation tab, build a new regression model, and re-analyze the data. It is very common to move backward and forward while experimenting with different model parameters. The interface allows the user to click on a previous tab at any time to try out new settings. Regression model analysis is often trial-and-error because the underlying model of the response behavior is unknown. The whole point of experimental design is to allow better understanding of the structure of the response.

Figure 9.15 shows the list of available transformations on the left side of the transformation panel. Selecting any transformation changes the graphic to the right of the transformation list. The first seven transformations belong to the power family. The graphic for each power transformation contains a sample of what the Residuals vs. Predicted Response plot should look like if the selected power transformation is applicable. All graphics contain information about when the selected transformation is most useful. Most transformations have restrictions on the input data, which are illustrated in each transformation’s graphic. Many transformations have optional parameters that can be entered in the text fields that appear below the transformation list when appropriate. The most common parameter is the translation constant \( c \). Adding the appropriate constant to each reward variable will translate the values to a range for which the selected transformation is valid. If the selected transformation is invalid for any response values, a dialog will warn about the offending values when the user tries to progress to the next analysis step. The ratio of maximum-to-minimum response values is presented below each graphic to aid in the decision on whether a transformation is necessary.

Most commonly, no transformation is applied to the response. The remaining analysis steps are identical regardless of whether the response has been transformed. If you do use a transformation, it is important for you to remember that regression model plots and prediction are presented in the transformed domain.
Whenever possible, the response is labeled wrapped in the corresponding transformation function. For example, if the natural logarithm transformation is used, the response will be labeled “Ln[response_name].”

9.8 Model Selection

After the appropriate transformation has been selected, the next step is to click on the Model tab and choose the effects to include in the regression model. The method for choosing effects varies depending on whether a factorial or a response surface design is being analyzed. Each case will be presented separately.

9.8.1 Factorial model selection

For factorial designs, one may select effects either graphically or from a table. Each selection interface has a separate tab panel within the model panel. The Normal Plots tab shows the normal quantile plot of effects. Figure 9.16 shows an example normal plot of effects. If the effects follow a normal distribution, then the normal quantile plot of ordered effects will follow a straight line. The outliers are the significant effects. On the normal plot of effects, outliers appear at the top and bottom of the plot. Figure 9.16 shows three outliers: two at the top and one at the bottom. To select an effect for inclusion in the regression model, the user simply clicks the point corresponding to the effect. The selected points will turn from blue to red, and the label of the chosen effect will be placed near the selected point. An ordinary least squares straight line will be drawn through all unselected blue points.

The most common graphical method for selecting effects is the half-normal plot. The half-normal plot is identical to the normal plot, but the absolute values of the effects are plotted. All outliers appear in the top right section of the plot. Figure 9.17 shows the same data as Figure 9.16, but plotted on a half-normal quantile plot. All three outliers (largest effects) are selected at the top right of the plot. In this example, main effects B, E, and G are included in the regression model.

You may switch back and forth from normal to half-normal views of the effects plots by selecting the appropriate button under the factor list at the left side of the normal plots panel. You can unselect all selected points by pressing the Clear Selection button. The Fill Points checkbox controls whether the plotted points are filled-in colored disks, or simply hollow circles. Deselecting the checkbox can make point labels easier to read on a very crowded plot.

The user can also select effects via the table in the Effects tab panel. Figure 9.18 shows the effects table corresponding to the data in the normal plots in
9.8. MODEL SELECTION

Figure 9.16: Normal plot of effects for factorial design.

Figure 9.17: Half normal effects plot.
Figures 9.16 and 9.17 Only the portion of the table listing the nonaliased effects is shown, in order to conserve space. The first column lists all effects up to five factor interactions. You can include in the regression model any effect that is not aliased by clicking on the corresponding checkbox in the In Model column. The next two columns list the effect estimate and sum of squares value for each effect. If an effect is aliased, the word “aliased” appears instead of a sum of squares value. The % Contribution column lists the percentage that the corresponding effect’s sum of squares contributes to the total sum of squares. The elements of the % Contribution column will always add to 100%, except in Plackett-Burman designs in which interactions contribute a nonzero sum of squares. Only main effects are listed for Plackett-Burman designs. The last column lists the aliases of the effects in the first column. Only aliases for up to five factor interactions are listed. A complete list of the alias structure can be found in the Summary tree node.

Each column in the effects table is sortable. Clicking on any column header will sort the list in descending order. Clicking again will sort in ascending order. The standard procedure is to click on the % Contribution column header to bring the effects that explain the most observed variability of the response to the top of the column. Effects with larger percent contributions are more likely to be significant regression model terms. Figure 9.18 shows that the top three effects, E, G, and B, are selected for inclusion in the regression model. Those effects were also selected in the normal plots in Figures 9.16 and 9.17. Selecting an effect from the effects table is equivalent to selecting the same effect from the normal plot.

The effects table in Figure 9.18 shows that effects E and G explain 52.7% and 33.3% of the observed variability of the response. Suppose that experience with the system being modeled indicates that the interaction EG should exert a strong effect on the response. Suppose also that factor B is not expected to be influential.
on the response. However, according to the effects table, B explains 13.9% of
the observed variability. Careful inspection of the aliases for effect B reveals that
B is aliased to EG (and several other interactions). This observation would lead
the experimenter to conclude that the 13.9% variability is due to interaction EG,
not factor B. Inclusion of factor B in the regression model would lead to invalid
conclusions. Fortunately, there is a way to alter the alias arrangement in the effects
table so that the correct effect can be added to the regression model.

Right-clicking on any row of the Effect or Aliases column activates a popup
menu allowing the aliases for the row’s effect to be edited. Figure 9.19 shows the
dialog that appears after a user has selected the popup menu item to edit the aliases
for effect B. The popup shows a pull-down menu listing the aliases of B. After term
EG has been selected, the effects table is updated as shown in Figure 9.20. Factors
E, G, and their interaction EG will then be used to build the regression model.

Regression models in the DOE study editor are required to be hierarchical. In a
hierarchical model, all parent effects for all interactions in the model must also be
included in the model. For example, if interaction ABC is included in a regression model, the parent effects A, B, C, AB, AC, and BC must also be included. Möbius automatically includes the terms necessary to make the selected model hierarchical. This automatic update is performed whenever the user switches to a different view in the main panel. Clicking from the Effects tab to the Normal Plots tab will automatically cause all parent effects to be added to the selected regression model.

Response surface model selection

The method of effect selection for response surface designs differs from the factorial method. There is no graphical procedure for selecting effects for response surface designs. Under the Model tab in response surface designs, there are two tabs to choose from: the Fit Summary and Model Selection tabs.

![Figure 9.21: ANOVA table in fit summary panel.](image)

<table>
<thead>
<tr>
<th>Source</th>
<th>Sum of Squares</th>
<th>DF</th>
<th>Mean Square</th>
<th>F Value</th>
<th>Prob &gt; F</th>
<th>Significant?</th>
<th>Aliased?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>538.24</td>
<td>1</td>
<td>538.24</td>
<td>851197</td>
<td>2.31E-9</td>
<td>Significant</td>
<td></td>
</tr>
<tr>
<td>Linear</td>
<td>6.6992</td>
<td>4</td>
<td>1.6348</td>
<td>0.2237</td>
<td>0.9211</td>
<td>Not Significant</td>
<td></td>
</tr>
<tr>
<td>2 Var Int</td>
<td>27</td>
<td>6</td>
<td>4.5</td>
<td>0.5327</td>
<td>0.7345</td>
<td>Not Significant</td>
<td></td>
</tr>
<tr>
<td>Quadratic</td>
<td>222.1417</td>
<td>4</td>
<td>52.4794</td>
<td>12.0968</td>
<td>0.0002</td>
<td>Significant</td>
<td></td>
</tr>
<tr>
<td>Cubic</td>
<td>15.5695</td>
<td>8</td>
<td>1.9461</td>
<td>7.9646</td>
<td>0.1168</td>
<td>Not Significant</td>
<td>Aliased</td>
</tr>
<tr>
<td>Residual</td>
<td>0.5</td>
<td>2</td>
<td>0.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>590</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The fit summary panel summarizes the fit of several regression models to the observed response values. An ANOVA table is used to calculate the significance of each model. Figure 9.21 presents an example ANOVA table for five models. The mean regression model contains only the average response value. If $\beta_0$ is the average, then the model is simply $y = \beta_0$. The linear model contains the mean and all main effects. The two-variable interaction model contains all effects from the mean and linear models, plus all two-factor interactions. Each successive model adds more terms to the model. The quadratic model adds pure second-degree effects ($A^2$, $B^2$, $C^2$...). The cubic model adds all pure cubics ($A^3$, $B^3$, $C^3$...), all mixed terms ($A^2B$, $A^2C$, $AB^2$...), and all three-factor interactions ($ABC$). The ANOVA is calculated using the sequential sum of squares method explained in Section 3.5. The Significant column in the ANOVA table tells whether the corresponding model’s additional terms added to the sequential model contribute significantly to the observed variability in the response. From Figure 9.21, it is clear that the pure quadratic terms contribute significantly to the regression model. The linear and two-variable interaction models are not significant. The response surface
9.8. MODEL SELECTION

must exhibit second-order characteristics, which are best modeled with quadratic regression polynomial terms.

A second table of model statistics presents further information to help users choose appropriate regression models. Figure 9.22 lists useful statistics for each of the models presented in the ANOVA table in Figure 9.21. The Adjusted $R^2$ column is the most important. The table exhibits the fact that the $R^2$ value always increases as terms are added to the regression model, but the adjusted $R^2$ can actually decrease. A negative adjusted $R^2$ for the linear and two-variable interaction models indicates that they are not adequate models of the response. The quadratic model explains 74.59% of the response’s observed variability, according to the adjusted $R^2$ value.

In both example fit summary tables, the cubic model is aliased. The concept of an aliased model in response surface methodology is similar to that in factorial designs. An aliased model is one in which there have not been enough experiments performed to estimate all of the effects in the model. In order to de-alias a response surface model, it is necessary to add more experiments in the design panel, as explained in Section 9.6.

![Figure 9.22: Fit summary model statistics table.](image)

The last piece of information presented in the fit summary panel is the sum of squares details for the ANOVA table. Each effect added to the sequential model has a corresponding sum of squares term. There are two separate sum of squares lists. The first is sorted by effect name; the second is sorted in descending order by sum of squares magnitude. The most influential effects will be placed at the top of the second list. It is important to remember that for a sequential sum of squares, the order in which effects are added to the model may affect the magnitude of the sum of squares. The sorted sum of squares list is useful for selecting potentially highly significant effects for inclusion in the regression model.

Effects are added to the regression model in the model selection panel. The table in Figure 9.23 lists each effect in the first column and a corresponding checkbox in the second column. A single mouse click on the checkbox toggles the effect’s in-
clusion in the model. When any row is selected, the user can activate a popup menu by right-clicking anywhere on the table. The popup menu, shown in Figure 9.23, contains items for removing all selected effects from or adding all selected effects to the regression model. An alternative to clicking directly on the table to select effects is to use the Model pull-down menu above the table. From the pull-down menu, the mean, linear, quadratic, cubic, or other model may be selected.

By default, the highest-degree effect listed in the model selection panel is three. You can alter this maximum degree by selecting a different maximum degree from the preferences dialog. The choices for the maximum degree are two through six. Another way to add a specific higher-order effect to the table is to type the desired effect in the text field at the top of the Model selection panel and click the Add Effect button. For example, if effect A5B is desired, the user may type any of “A5B”, “a5b”, “A5B”, or “a5B” and push the Add Effect button to place $A^5B$ in the table. All parent effects required to support hierarchy are automatically added to the table. Therefore, adding $A5B$ will automatically add $A^4$, $A^3B$, $A^5$, and $A^4B$ to the table if they are not already there. The maximum degree effect allowed is six. For example, $AB^2C^3$, $ABCDEF$, and $A^6$ are all sixth-degree effects. To remove any user-added effects, you can press the Remove Effect button; a dialog will prompt you to select from a list of added effects. The selected effect and all
9.8. MODEL SELECTION

parent effects required for hierarchy are then removed from the table.

By default, no exponents of degree four or higher are displayed in superscript font. If the computer system running the Möbius interface has the superscript fonts installed, then you can activate their display by clicking the **Use Exponent Font in Effects** checkbox in the preferences dialog. Selecting this option also affects the display of exponents in the fit summary and ANOVA panels.

9.8.2 ANOVA

After the effects for the regression model have been selected, a statistical analysis can be performed to evaluate the quality of the model. Clicking on the **ANOVA** tab reveals a wide variety of statistical output about the regression model. Figure 9.24 shows the first four pieces of information from an example response surface design. The first two items summarize the response and transformation information. The next two items do not always appear. If effects had to be automatically added to the regression model in order to make it hierarchical, then a list of those effects will be presented below the transformation data, as presented in Figure 9.24. The final piece of data applies only to response surface designs. It is possible in the model selection panel to select more effects than can be independently estimated given the available degrees of freedom. If this happens, some of the selected effects will be aliased to other effects in the model. As effects are sequentially added to the regression model from lower to higher order, any effect that is aliased to an effect already in the model is not actually added to the model. Instead, it is placed in the list of aliased terms presented in the **ANOVA** panel. The alias structure is dependent on which effects are already in the model. Therefore, the complete alias structure for the selected model is presented in the **ANOVA** panel after the entire model is selected.

The next item in the **ANOVA** panel is the analysis of variance table. Figure 9.25 shows a screen capture of the ANOVA table for a typical Möbius model. The significance of the model and each effect should be analyzed. If an effect is not significant and it is not required to support the model hierarchy, then it should be removed from the model. The significance and insufficiency thresholds are defined in the preferences dialog. The default thresholds are 0.05 for significant and 0.1 for insignificant. Any effect with a P-Value less than the significance threshold is significant. Any effect with a P-Value greater than the insufficiency threshold is not significant. The significance of an effect with a P-Value between the two thresholds is left to the experimenter to decide.

A summary of the ANOVA results is presented after the ANOVA table. The summary in Figure 9.25 gives an interpretation of the model P-Value. Multiplying the model P-Value by 100 gives the percent possibility that the given significance
Figure 9.24: Initial data in ANOVA panel.

Figure 9.25: ANOVA table with summary.
9.8. MODEL SELECTION

level could occur strictly by chance. A summary of which terms are significant and not significant is also given.

After the ANOVA summary, there are two tables of regression model statistics. The theory behind these statistics was presented in Section 3.6. Figure 9.26 shows a sample with a summary of the data below the tables. The adjusted R2 should be as close to one as possible. The model should explain as much variability as possible without insignificant effects. The prediction capability of the regression model is best when the adjusted R2 and prediction R2 are as close to each other as possible. If the two are within 0.20 of one another, then the predictive quality is acceptable.

<table>
<thead>
<tr>
<th>Standard Deviation</th>
<th>Mean</th>
<th>Coefficient of Variation</th>
<th>PRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>5.25</td>
<td>9.5338</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R'</th>
<th>Adjusted R'</th>
<th>Prediction R'</th>
<th>Adequate Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9468</td>
<td>0.9314</td>
<td>0.8431</td>
<td>12.7279</td>
</tr>
</tbody>
</table>

The Adjusted R2 shows that the model explains 93.14% of the observed variability.
The Prediction R2 is within 0.20 of the Adjusted R2, implying good predictive capability of the model.
The Adequate Precision is greater than 4, indicating that the model's signal-to-noise ratio is adequate for exploring the design space.

Figure 9.26: Regression model statistics table.

A table of diagnostic statistics for each experiment is presented next. The data in each column are presented graphically in the Diagnostic Plots panel. The tabular presentation is an alternative to the scatter plots and is preferable for some users. For experiments with leverage values of one, the student’s residuals, Cook’s distance, and outlier ε are undefined.

Below the diagnostic statistics table are two representations of the regression model polynomial. Figure 9.27 shows both representations. Each polynomial is written with one term per line to facilitate formatting. For example, the first equation in Figure 9.27 is

\[ \text{Performance.mean} = 5.25 + 1.5A + 0.75B + 0.5AB \] (9.1)

The first equation is presented in coded global variable terms. The coded variables are the natural values scaled from –1 to +1. They are the values used for internal calculations. The equation in terms of natural global variable values is the one used to predict response values. The first item in the Design menu allows the natural
CHAPTER 9. DOE STUDY EDITOR INTERFACE

Figure 9.27: Regression equations in coded and natural formats.

Form of the regression equation to be exported to a file. The file may be in either plain text or Mathematica notebook format.

The final piece of data presented in the ANOVA panel is a table of standard errors and confidence intervals for each regression model coefficient. Figure 9.28 shows an example table. The standard error is the standard deviation of the coefficient estimate. The 95% confidence intervals represent the range in which the coefficients should lie 95% of the time. If the range contains zero, then the coefficient is not significant.

Figure 9.28: Standard errors and confidence intervals for regression coefficients.

The area of the ANOVA panel containing the table of standard errors is an editable text area. Comments can be entered in the text area. The entire ANOVA panel can be exported to an HTML file using the Design menu’s Export Text item. A plain-text version of the ANOVA table can also be appended to the ANOVA panel output if the appropriate checkbox is selected in the preferences dialog. The plain-text version is useful for copying and pasting text into the notes panel or to
9.9. **Diagnostic Plots**

If the regression model produced satisfactory results in the ANOVA panel, the analysis may proceed to the **Diagnostic Plots** panel. The diagnostic plots consist of six scatter plots of the data from the Diagnostic Statistics table in the ANOVA panel. Six buttons on the left side of the **Diagnostic Plots** panel control which of the six plots is displayed. Figure 9.16 shows the panel with an example normal plot of studentized residuals. Each point in the diagnostic plots can be selected with a single mouse click. An information panel above the buttons gives details about the selected point, such as the coordinates and the corresponding experiment number.

![Diagnostic Plots Panel](image)

**Figure 9.29:** Normal plot of studentized residuals.

The normal plot of studentized residuals in Figure 9.29 shows that the response residuals generally follow a straight line throughout the central portion of the plot. Therefore, it can be assumed that the residuals are normally distributed. The outliers at the lower and upper extremes are not sufficient grounds for dismissal of the normality assumption. The outlier in the upper right portion of the plot is selected in red. The information panel reveals that the selected outlier corresponds to experiment 45.

Clicking on the *Residuals vs. Predicted* button reveals the plot in Figure 9.30. With the exception of the selected outlier, no visible trend is present. Therefore, the assumption of constant variance of residuals is valid. As in the previous plot,
the outlier corresponds to experiment 45. A checkbox below the six plot selection buttons toggles the display of studentized residuals versus ordinary least squares residuals. The studentized form of residuals is preferred. However, in some cases, the studentized residuals are not defined. In those cases, the ordinary residuals must be used to validate the assumptions.

![Figure 9.30: Student residuals vs. predicted response diagnostic plot.](image)

The Predicted vs. Actual plot in Figure 9.31 shows that most points are gathered near the 45° line, implying a fairly good least squares fit. The point farthest from the line is selected in red and, again, corresponds to experiment 45. The data gathered from the first three plots indicate that experiment 45 may have a problem.

The outlier $t$ plot in Figure 9.32 shows that experiment 45 is indeed an outlier. The information panel indicates in red type that the value of the outlier $t$ for experiment 45 is barely larger than 3.5, making it an outlier. Further investigation by the experimenter would be required in order to either justify or discount the data from the outlying experiment.

The leverage plot in Figure 9.33 shows a pair of points with leverage values greater than twice the average leverage value. The two points exert a disproportionately large influence on the model fit. The Cook’s distance plot in Figure 9.34 does not reveal any evidence of disproportionately large influence among the experimental points. Any points that cross the threshold lines in the leverage or Cook’s distance plots should be investigated.

If all diagnostic plots look good, then the analysis may proceed to the inspection of the regression model graphs. If any of the diagnostic plots reveal a problem,
9.9. DIAGNOSTIC PLOTS

Figure 9.31: Predicted vs. actual diagnostic plot.

Figure 9.32: Outlier t diagnostic plot.
then the analysis should not proceed to the next step. Instead, transformations, model reduction, or additional experiments should be considered. Conclusions from the Model Plots and Prediction phases of the analysis are valid only if all diagnostic tests show satisfactory results.

![Figure 9.33: Leverage diagnostic plot.](image)

After verifying that the regression model is adequate for prediction, the user can use plots of the regression model to analyze response behavior. There are five different types of plots available in the model plots panel. The user selects the type of plot by clicking on the appropriate tab within the model plots panel. All of the plot panels have the same layout. Figure 9.35 shows the layout of the one-factor plot panel. The plot occupies the right side of the panel. The left middle portion of the panel contains slider controls for each global variable in the regression model. A slider can be enabled or disabled. If a slider is disabled, it will appear dimmed. In Figure 9.35, the slider for global variable A:Memory_RAM is disabled, but the other two sliders are enabled. A slider becomes disabled when its corresponding global variable is selected as an axis label. In Figure 9.35, the horizontal, or x-axis, is labeled as A:Memory_RAM. Axes are selected from pull-down menus at the bottom left corner of the main panel. An information panel above the sliders displays the axes labels and the current values of all factors corresponding to all enabled sliders. In Figure 9.35, the values for global variables B and C are listed in the information panel at their lowest values of 1 and 300, which correspond to the selected values of each enabled slider. As the sliders are adjusted, the plot and information panels update immediately. This allows for “real time” feedback of
9.9. DIAGNOSTIC PLOTS

Figure 9.34: Cook’s distance diagnostic plot.

Figure 9.35: One Factor model plot.
response behavior as the global variable values are varied. The decimal format for the numbers in all model plots can be changed in the preferences dialog. The default number format is #.##, which limits the number of digits to the right of the decimal point to two.

The first type of model plot is the one-factor plot, presented in Figure 9.35. The one-factor plot shows how the response behaves as a single global variable is varied from its low to high value. The example plots in this section are from the workstation example presented earlier. Figure 9.35 shows how workstation performance increases as the amount of RAM is increased from 1 to 16 GB, with 1 processor and 300 GB of hard disk space. The Points Per Line text field below the sliders allows the resolution of the plot to be adjusted. Users draw all plots by drawing straight lines connecting a given number of sample plot points. Increasing the points per line makes the graph smoother. For factorial designs, all one-factor plots will be straight lines, so increasing the points per line does not affect the plot. In response surface regression models, global variables can be quadratic and higher-degree, which can add curvature to the one-factor plot.

The interaction plot is useful for revealing how regression model global variables act together on a response. Figure 9.36 shows an example interaction plot. Two lines of workstation performance are plotted versus the amount of RAM: one line with one processor, the other with four processors. If the number of processors and the amount of RAM do not affect each other, then the lines will be parallel. However, Figure 9.36 reveals that the lines are not parallel, indicating that an interaction is present between the two global variables. Performance increases more sharply with four processors than with one as the RAM is increased from 1 to 16 GB. You can vary the hard disk size by adjusting its slider to see if it affects the interaction.

If there are \( m \) factors in the regression model, then the response can be represented as a surface in \( m \)-space. Any two of the \( m \) factors can be selected as the axes for a graph of a two-dimensional projection of the response. The contour plot helps the user visualize the shape of the response using a two-dimensional projection. Figure 9.37 shows a contour plot of workstation performance versus the number of processors and the amount of RAM. Contours mark lines of constant performance. Each contour line is a different color. A label with the same color indicating the value of the response is placed near each contour line. Figure 9.37 shows how performance increases from 3.7 on the left side of the plot to 7.75 at the upper right corner. Checkboxes for toggling the display of the contour labels and the display of a grid are available. The grid size is determined by the number of points per line. The number of contours can be adjusted in the text field above the axes’ pull-down menus.

The last two types of plots are useful visuals for revealing how the response
Figure 9.36: Interaction model plot.
Figure 9.37: Contour model plot.
varies as two or three global variables vary from their low to high values. The square plot and cube plot in Figure 9.38 show how workstation performance varies as each global variable is varied from one extreme to the other. The cube plot indicates that the size of the hard disk has a much smaller effect on performance than the other two global variables do.

**9.10 Prediction**

If the model plots show that the regression model is acceptable, the model can be used to predict the response at exact points in the design space. The prediction panel provides a method for selecting a point in the design space and calculating the response value, standard errors, confidence intervals, and prediction intervals for the selected point. Figure 9.39 shows the interface for selecting the prediction point. Each global variable in the regression model is listed at the top of the prediction panel with a corresponding slider control and text field. The values in the text fields represent the coordinates in the design space. You can alter the text field values by adjusting the slider controls from the low to high values. You may also type values directly into each text field. After entering a value in the text field, push the **Evaluate** button to update the labels for the prediction point data. A text area at the bottom of the prediction panel displays a summary of information about the global variables in the study. Whenever the **Append** button is pushed, the data for the predicted point are appended to the text area. The text area is editable. The **EXPORT TEXT** item in the **Design** menu allows the entire contents of the text area to be exported to a text file.

The decimal format can be changed in the preferences dialog. The default format is to display five digits to the right of the decimal point. The confidence levels are based on the significance threshold defined in the preferences dialog. Be cautious about predicting response values that are far from the design center. Any global variable value that is outside the original design space will be highlighted in red in the corresponding text field, and any appended text in the text area will be marked with an asterisk.
Figure 9.38: Square and Cube model plots.
Figure 9.39: Prediction panel.
Part III

Solving Models with Möbius
Chapter 10

How to pick the solver

Möbius provides two types of solvers for obtaining solutions on measures of interest: simulation and numerical solvers. The choice of which type of solvers to use depends on a number of factors. More details on these factors are provided in the chapters on simulation (Chapter 12) and numerical solvers (Chapter 13).

In general, the simulation solver can be used to solve all models that were built in Möbius, whereas numerical solvers can be used on only those modes that have only exponentially and deterministically distributed actions. In addition, simulation may be used on models that have arbitrarily large state-space descriptions, whereas numerical solvers are limited to models that have finite, small state-space description (that may be held in core memory). Furthermore, simulation may be more useful than numerical solvers for stiff models.

On the other hand, all numerical solvers in Möbius are capable of providing exact solutions (up to machine precision), whereas simulation provides statistically accurate solutions within some user-specifiable confidence interval. The desired accuracy of the computed solutions can be increased without excessive increase in computation time for most numerical solvers, while an increase in accuracy may be quite expensive for simulation. Additionally, full distributions may be computed for results from the numerical solvers, but usually not for results from simulation. Furthermore, for models in which numerical solvers are applicable, detection of rare events incurs no extra costs and requires no special techniques, whereas such computation by simulation is extremely expensive and uses the statistical technique of importance sampling.
Chapter 11

Transformers

11.1 Introduction

Some of the solution techniques within Möbius, such as the simulator, operate directly on the model representation defined using the Atomic and Composed editors described in earlier chapters of the manual. These solvers operate on the model using the Möbius model-level abstract functional interface. There are other solution techniques, specifically the numerical solvers described in the next chapter, which require a different representation of the model as an input. Instead of operating on the high-level model description, numerical solution techniques use a lower-level, state space representation, namely the Markov chain.

11.2 Flat State Space Generator

The flat state space generator\(^1\) is used to generate the state space of the discrete-state stochastic process inherent in a model. The state space consists of the set of all states and the transitions between them. Once the state space has been generated, an appropriate analytical solver can be chosen to solve the model, as explained in Chapter 13. While simulation can be applied to models with any underlying stochastic process, numerical solution requires that the model satisfy one of the following properties:

1. All timed actions are exponentially distributed (Markov processes).

2. All timed actions are deterministic or exponentially distributed, with at most

---

\(^1\) This was the only state space generator (SSG) available prior to version 1.6.0 and it was simply called the state space generator. From that version on, it is called the flat state space generator.
one deterministic action enabled at any time. Furthermore, the firing delay of the deterministic actions may not be state-dependent.

The only restrictions on the use of instantaneous (zero-timed) actions are that the model must begin in a stable state, and the model must be stabilizing and well-specified \[28\]. It should also be noted that the reactivation predicates (see Section 5.1.1) must preserve the Markov property. In other words, the timed actions in the model must be reactivated so that the firing time distributions depend only on the current state, and not on any past state. That rule pertains only to timed actions with firing delays that are state-dependent.

The flat state space generator consists of a window with two tabs, SSG Info and SSG Output, which are discussed below.

11.2.1 Parameters

The SSG Info tab (Figure 11.1) is presented when you open the interface, and allows you to specify options and edit input to the state space generator.

![Figure 11.1: Flat state space generator input.](image)

- The Study Name text box specifies the name of the child study. This box will show the name of the study you selected to be the child when you created the new solver. You can change the child study by typing a new name in the box or clicking the Browse button and selecting the solver child from the list of available studies.

- The Experiment List box displays the list of active experiments in the study, and will be updated upon any changes made through the Experiment Activator button.
11.2. FLAT STATE SPACE GENERATOR

- The **Experiment Activator** button can be used to activate or deactivate experiments defined in the child study, and provides the same functionality as the study editor Experiment Activator described in Section 8.3.

- The **Run Name** text box specifies the name for a particular run of the state space generator. This field is used to name trace files and output files (see below), and defaults to “Results” when a new solver is created. If no run name is given, the solver name will be used.

- The **Build Type** pull-down menu is used to choose whether the generator should be run in Optimize or Normal mode. In normal mode, the model is compiled without compiler optimizations, and you can specify various levels of output by changing the trace level, as explained below. This mode is useful for testing or debugging a model, as a text file named `<Run Name>_Exp<exp>_<trace.txt` will be created for each experiment and contain a trace of the generation. In optimize mode, the model is compiled with maximum compiler optimizations, and all trace output is disabled for maximum performance. Running the generator will write over any data from a previous run of the same name, so change the **Run Name** field if you wish to save output/traces from an earlier run.

- The **Trace Level** pull-down menu is enabled only when Normal mode is selected as the Build Type. This feature is useful for debugging purposes, as it allows you to select the amount of detail the generator will produce in the trace. The five options for Trace Level are:
  
  **Level 0: None**  No detail in the trace. This would produce the same output as if optimize mode were selected.

  **Level 1: Action Name**  For each new state discovered, prints the names of the actions enabled in this state. These are the actions that are fired when all possible next states from the current state are being explored.

  **Level 2: Level 1 + State**  The same as Level 1, with the addition that the current state is printed along with the actions that are enabled in the state.

  **Level 3**  has not yet been implemented in Möbius.

  **Level 4: All**  The same as Level 2, with the addition that the next states reachable from each state currently being explored are also printed.

While a higher trace level will produce more detail in the generator trace and aid in debugging a model, it comes at the cost of slower execution time and larger file size.
• The Hash Value text box is used by the hash table data structure, which stores the set of discovered states in the generation algorithm. This number specifies the point at which the hash table is resized. Specifically, it is the ratio of the number of states discovered to the size of the hash table when a resizing occurs. The default value of 0.5, which means that the hash table is resized when it becomes half full, should suffice for most purposes.

• The Flag Absorbing States check box, when selected, will print an alert when an absorbing state is encountered. An absorbing state is one for which there are no next (successor) states.

• The Place Comments in Output check box, when selected, will put the user-entered comments in the generator output/trace. If the box is not checked, no comments will appear, regardless of whether any have been entered.

• The Edit Comments button allows you to enter your own comments for a run of the generator. Clicking this button brings up the window shown in Figure 11.2. Type any helpful comments in the box and hit OK to save or Cancel to discard the comments. The Clear button will clear all the text from the comment box.

Figure 11.2: Edit Comments window.

• The Place Comments in Output check box, when selected, will put the

---

2 A resizing rehashes and roughly doubles the size of the hash table.
11.2. FLAT STATE SPACE GENERATOR

user-entered comments in the generator output/trace. If the box is not checked, no comments will appear, regardless of whether any have been entered.

- The Experiment List box displays the list of active experiments in the study, and will be updated upon any changes made through the Experiment Activator button.

11.2.2 Generation

Clicking the Start State Space Generation button switches the view to the SSG Output tab and begins the process of generating the state space. The output window appears in Figure 11.3

![Flat state space generator output.](image)

First the models\(^3\) are compiled and the generator is built. For each active experiment, the values of the global variables are printed; if normal mode is being used, information about the trace level and hash table is also displayed. Then the state space is generated for the experiment. While it is being generated, the States Generated box and progress bar at the bottom of the dialog indicate the number of states that have been discovered. This number is updated once for every 1,000 states generated and when generation is complete. At any time, you may use the Stop button to terminate the run. A text file named `<Run Name>_output.txt` will be created. It will mirror the data output in the GUI.

---

\(^3\)Here models refers to the atomic/composed models, performance variable, and study.
A final remark is in order about the notion of state during generation of the state space of the stochastic process. In Möbius, a state is defined as the values of the state variables along with the impulse reward (see Section 7.1.1) associated with the action whose firing led to that state. Therefore, for a model on which impulse rewards have been defined, if two different actions may fire in a given state and lead to the same next state, they give rise to two unique states in the state space. The reason is that when impulse rewards are defined, the state space generator must store information about the last action fired. The result is a larger state space for models on which impulse rewards have been defined. It is important to realize this when analyzing the size of the generated state space.

11.2.3 Command line options

The flat state space generator is usually run from the GUI. However, since the models are compiled and linked with the generator libraries to form an executable, the generator can also be run from the command line (e.g., from a system shell/terminal window). The executable is found in the directory

```
<Möbius Project Root>/<Project Name>/Solver/<Solver Name>/
```

and is named `<Solver Name>Gen_<Arch>[_debug]`, where Arch is the system architecture (Windows, Linux, or Solaris) and _debug is appended if you build the flat state space generator in normal mode. The command line options and their arguments are as follows:

- **-P experimentfilepath**
  Place the created experiment files in the directory specified by `experimentfilepath`. This argument can be a relative path. By default, the experiment files will be placed in the current directory, but you can specify a path using the option “-E.”

- **-N experimentname**
  Use `experimentname` to name the experiment files created. These files will have extensions .arm or .var, and will be placed in the directory specified by the “-E” option.

- **-B experimentnumber**
  Generate the state space for the experiment numbered `experimentnumber`, with the first experiment having number 0. For example, use “-B1” to run the second experiment (e.g., Experiment_2). Defaults to 0.
11.3 SYMBOLIC STATE SPACE GENERATOR

-h hash_value
Use hash_value as the hash value described above. Defaults to 0.5.

-a
Flag absorbing states.

-F filetype[1-6]
Use filetype as the format for the experiment file. The only filetype currently supported in the flat state space generator is “-F1,” or row ASCII format, which is also the default option. Other file formats may be available in future versions.

-e, -s db_entity, db_macserver
These are internal options for interaction with the database. You can safely ignore them.

-l trace_level[0-4]
Use trace_level as the trace level described above. Only applies when you are running the debug executable.

-t tracefilename
Create a trace file named tracefilename as output. If this option is not specified, output is directed to the standard output stream.

Example: ssgGen Windows_debug -E. -B0 -l4 -ttrace.txt

- <Solver Name> is ssg
- <Arch> is Windows

This command will generate the state space for the first experiment (Experiment_1) with trace level 4, place the experiment files in the current directory, and create a trace file named trace.txt.

11.3 Symbolic State Space Generator

The symbolic state space generator (symbolic SSG) is a component that was added to the Möbius tool for version 1.6.0. In this chapter, we first introduce this component, discuss its advantages and limitations, and describe how to use it in an efficient way. Then, we will explain how to fill up each of the input parameters of its corresponding GUI editor. The different phases of the state space generation will be briefly described. Finally, we will provide instructions on how to run the
symbolic SSG from the command line. For more details on the inner workings of the symbolic SSG, please see [13, 14].

11.3.1 Introduction

In solving a model using numerical solution, a modeler must first run a state space generator on the model to build a representation of the stochastic process underlying the model, and then use a numerical solver to compute the measures (s)he is interested in.

A numerical solution algorithm needs data structures to represent two essential entities in the memory: the stochastic process (input) and one or more probability distribution vectors (output). The stochastic process is generated by a state space generator. In this chapter, we only consider stochastic processes that are continuous time Markov chains (CTMCs), and can therefore be represented by state transition rate matrices whose entries are the rates of exponentially distributed timed transitions from one state to another.

In many numerical solvers, sparse matrix representation and arrays are respectively used to represent the process and the probability vector(s). In such a design of data structures, the stochastic process represented by a sparse matrix almost always takes more memory space than the probability vectors.

A problem arises when the model under study is so large that the memory requirement of the two data structures exceeds the amount of memory available on a computer. For example, to analyze a model with ten million states for a specific point of time using the transient solver (TRS) and the data structures mentioned above, at least $10^7 \times 10 \times 8$ bytes $= 800$ Mb to $10^7 \times 20 \times 8$ bytes $= 1600$ MB (depending on the model) of RAM are needed, assuming that each state leads to 10 to 20 other (successor) states and each floating-point number requires 8 bytes of storage.

That problem has stimulated a lot of research looking for more “practical” data structures, i.e., more compact (with smaller space requirements) data structures that do not incur too much time overhead. The most practical data structure that virtually all numerical solvers (including ones in Möbius) use to represent probability distribution vectors is simply an array. Efforts by researchers to design better ones have failed so far.

However, there has been a lot of successful research on design of data structures for the representation of the state transition matrix. In particular, symbolic data structures (hence the name symbolic SSG) are widely used nowadays to store state transition matrices in a very compact manner. As opposed to sparse matrix representation, symbolic data structures do not have space requirements proportional to the number of nonzero entries. In virtually all cases, they take up orders of
magnitude less space than their sparse representation counterparts. This extremely low space requirement enables the modeler to numerically solve models that are much larger than was previously possible. For example, to do the same analysis we mentioned above, we only need about 100 MB of RAM using symbolic data structures. Because of the concept behind symbolic data structures and the way they are constructed, we see their benefits only in composed models, especially those consisting of atomic models of appropriate sizes; more information on this will be provided later in the chapter.

The major price a modeler has to pay is overhead in speed. Based on our measurements, when the symbolic SSG is used to generate the state space, numerical solution is expected to be about 3 to 10 times slower than it would be for the flat SSG. Notice that slowdown is not related to the state space generation itself. In fact, in virtually all cases, the symbolic SSG is faster than the flat SSG; in some cases it may be several times faster.

The symbolic SSG utilizes three types of symbolic data structures to represent the required entities. It uses MDDs (Multi-valued Decision Diagrams) for the set of reachable states, MxDs (Matrix Diagrams) for the state transition rate matrices, and MTMDDs (Multi-Terminal MDDs) for the values of the performance variables in each reachable state.

The symbolic SSG utilizes two types of lumping algorithm. Lumpining is a technique in which states of a CTMC that are behaviorally “equivalent” are lumped together and replaced by a single state. Therefore, the number of states of a “lumped” CTMC is always equal to or smaller than the number of states of the original CTMC. The performance variables computed from a lumped CTMC are exactly the same as the performance variables computed from the original one. That means that we save on both time and space if we solve a smaller lumped CTMC rather than the original one. For example, models that consist of a number of equal parts or components have a potential for lumping that reduces the size of the state space significantly.

The first lumping technique is called model-level lumping, in which the symmetry of a Replicate/Join model is exploited as explained in Section 6.1.1 to reduce the size of the state space. That lumping technique has been part of the symbolic SSG from the beginning.

The second lumping technique that has been added to symbolic SSG in version 1.8.0 is called compositional lumping. In compositional lumping, the lumping algorithm is applied to individual components (atomic models in the Möbius context) of a compositional model, whereas in state-level lumping the algorithm is applied to the overall CTMC generated from the model. In fact, that is the main advantage of the compositional lumping technique over state-level lumping; it is applied on CTMCs of components that are much smaller than the CTMCs of the whole
CHAPTER 11. TRANSFORMERS

model. That advantage means that, in general, compositional lumping algorithms require less time and memory space for their operation. Möbius supports ordinary compositional lumping. See [14] for more information.

To summarize, the symbolic SSG uses compact data structures to represent the state transition rate matrix of Markov chains such that it devotes most of the available RAM to the storage of the probability distribution vector(s). If possible, it uses model-level and compositional lumping algorithms to reduce the size of the state space of the CTMC that needs to be solved in order to compute the desired performance variables. That makes it possible for the modeler to numerically analyze models that are orders of magnitude larger than was previously possible using flat SSG and sparse matrix representation.

11.3.2 Limitations

As we mentioned above, speed is the major price we pay in using symbolic data structures. Moreover, there are a number of conditions that have to be satisfied before we can use the symbolic SSG:

- The stochastic process underlying a model has to be Markov. Therefore, only exponential and immediate (i.e., zero-timed) transitions are allowed.

- If the model under study is a composed one, immediate transitions must not affect or be affected by shared state variables.

- The reward model must not have impulse reward definitions.

- The starting state has to be stable (not vanishing), i.e., no immediate transition should be enabled in the starting state.

If any of the above conditions is not met for a model, the symbolic SSG is not usable, and its GUI shows a message indicating the unmet condition(s).

Finally, when using the iterative steady state solver, it is not possible to use the SOR (Successive Over Relaxation) variant with state spaces that are generated using symbolic SSG. Instead, the Jacobi variant should be used. The reason is that the symbolic data structure constructed by the SSG does not provide the elements of the state transition matrix in any specific order (see section 13.3.3).

11.3.3 Tips on efficient use

The time and space efficiency of the symbolic SSG can vary widely depending on the way the model is built. We have observed up to one order of magnitude of speed difference among a number of composed models that result in the same underlying
11.3. SYMBOLIC STATE SPACE GENERATOR

Markov chain. Theoretically speaking, even larger differences are possible. Here are two guidelines that will help modelers construct their models in a way that increases the performance of the symbolic SSG:

- *Do not* use symbolic SSG for atomic models, especially large atomic models. Not only does it not give any better performance, it drastically reduces both time and space efficiency of state space generation and numerical analysis. For those types of models, simply use the flat state space generator.

- Try to decompose the complete model into atomic models such that each of them has roughly 3 to 50 states. In the general case, it is impossible to estimate how many states an atomic model has. However, the modeler can sometimes make an estimation. Above 50 states, the larger the atomic models’ state spaces are, the more the state space generation performance will be negatively affected.

11.3.4 Parameters

The SSG Info tab (Figure 11.4) is presented when you open the interface, and allows you to specify options and edit input to the symbolic state space generator.

- The Study Name text box specifies the name of the child study. This box will show the name of the study you identified as the child when you created the symbolic SSG. You can change the child study by typing a new name in the box or clicking the Browse button and selecting the solver child from the list of available studies.

- The Experiment List box displays the list of active experiments in the study, and will be updated in response to any changes made through the Experiment Activator button.

- The Experiment Activator button can be used to activate or deactivate experiments defined in the child study, and provides the same functionality as the study editor Experiment Activator described in Section 8.3.

- The Run Name text box specifies the name for a particular run of the symbolic state space generator. This field is used to name trace files and output files (see below), and defaults to “Results” when a new symbolic SSG is created. If no run name is given, the symbolic SSG name will be used.

- The Build Type pull-down menu is used to choose whether the generator should be run in Optimize or Normal mode. In normal mode, the
model is compiled without compiler optimizations, and you can specify various levels of output by changing the trace level, as explained below. This mode is useful for testing or debugging a model, as a text file named <Run Name>_Exp<x>_trace.txt will be created for each experiment and contain a trace of the generation. In optimize mode, the model is compiled with maximum compiler optimizations, and all trace output is disabled for maximum performance. Running the generator will write over any data from a previous run of the same name, so change the Run Name field if you wish to save output/traces from an earlier run.

- The Trace Level pull-down menu is enabled only when Normal mode is selected as the Build Type. This feature is useful for debugging purposes, as it allows you to select the amount of detail the generator will produce in the trace. The two options for Trace Level are:

  **Level 0: None** The processing time for each phase of the generation is printed out (see the next section for more on generation phases). More-
over, the numbers of unlumped and lumped states are also shown. The output is the same as it would be if optimize mode were selected.

**Level 1: Verbose** This includes everything printed in Level 0, plus detailed information about the symbolic data structures, such as the number of levels, the size of the nodes in each level, the final number of nodes, the maximum number of nodes during the runtime, and the number of firings of local and global actions.

The symbolic SSG does not produce a great deal of debugging information; therefore, you will not notice much difference between the execution times of the two different trace levels.

- The numbers entered in the MDD Unique Table Size and MxD Unique Table Size text boxes are used by the MDD and MxD data structures. They are the sizes of the hash tables used to store all the unique nodes of the MDD and MxD data structures, respectively. Their default values are 50,000 and 10,000, and their minimum values are 5,000 and 1,000, respectively.

  There is no clear-cut set of rules on how to set these numbers. The default values should be appropriate for most models. For extremely large models for which the state space generation takes much more than an hour, double or quadruple the default values. Increasing the values above a certain value deteriorates the performance of the SSG algorithm.

- The Enable Compositional Lumping check box, when selected, activates the compositional lumping algorithm, which for some models (but not all of them) reduces the size of the CTMC that needs to be solved. Although enabling the algorithm increases the running time of the state space generation, in some cases, it leads to significant reduction in the numerical solution time.

- The Place Comments in Output check box, when selected, will put the user-entered comments in the generator output/trace. If the box is not checked, no comments will appear, regardless of whether any have been entered.

- The Edit Comments button allows you to enter your own comments for a run of the generator. Clicking this button brings up the window shown in Figure [11.2] Type any helpful comments in the box and hit OK to save or Cancel to discard the comments. The Clear button will clear all the text from the comment box.
11.3.5 Generation

Clicking the Start State Space Generation button switches the view to the SSG Output tab and begins the process of generating the state space. Snapshots of the output window in different phases of the generation appear in Figure 11.5.

![Snapshot (a) Just after starting.](image)
![Snapshot (b) Performing phase 6 (compositional lumping is enabled).](image)

Figure 11.5: Symbolic state space generator output.

As for the flat state space generator, the generation begins by compiling the model and building the generator. For each active experiment, the values of the global variables are printed.

The state space is then generated for the experiment in six phases if compositional lumping is enabled, and in five phases otherwise. Two progress bars at the bottom of the dialog show the amount of completed work. The Overall progress bar indicates which phase is in progress and the overall progress of the generation, and the Phase progress bar indicates the progress of each phase of the generation. The amount of time taken to complete each phase is printed after the phase is finished. At any time, you may use the Stop button to terminate the run. A text file named `<Run Name>_output.txt` will be created. It will mirror the data output in the GUI.

The 6 phases are:

1. The unlumped state space is generated in the form of an MDD data structure in a number of iterations (see Section 6.1.1 for definitions of lumped and unlumped state spaces). The total number of iterations depends on the model and is not known in advance. After each iteration is complete, the Phase

---

4Here *models* refers to the atomic/composed models, performance variable, and study.
11.3. SYMBOLIC STATE SPACE GENERATOR

progress bar shows the iteration number and the number of un lumped states that have been explored so far. When this phase is completed, the total num ber of explored unlumped states is printed. Phases 1, 3, and 6 are often the most time-consuming phases of the generation process.

2. The MxD data structure representing the state transition rate matrix of the unlumped CTMC is built in this phase. This phase is usually completed very quickly.

3. In this phase and the following one, lumping algorithms are applied on the state space to reduce the size of the CTMC that needs to be solved to compute the desired performance variables. In this phase, the compositional lumping algorithm is applied on the unlumped state space. The size of the state space lumped by the compositional lumping algorithm is printed out at the end of the step. This phase is performed in two subphases. In the first one, the initial partition for each atomic model is computed based on the values of the performance variables. In the second subphase, which is based on an efficient partition-refinement algorithm [14], the initial partition is refined repeatedly to compute the lumped state space for each atomic model. Virtually for all models, the second subphase is much faster than the first subphase.

4. In this phase, the model-level lumping induced by Replicate/Join operators is performed. The size of the state space after compositional and model-level lumping is printed out at the end of this step. This phase usually takes the least amount of time of all the phases.

5. For each of the performance variables defined on the model, one MTMDD symbolic data structure is built. This data structure gives the value of that performance variable for each state of the lumped state space.

6. The “mapping” MDD, a special type of the MDD data structure, is constructed in this phase [13]. This phase has two subphases. In the first, the set of states of the mapping MDD is computed; during this process, the progress bars are not updated because the cardinality of the set is not known yet. In the second subphase, the mapping MDD construction actually starts. While the mapping MDD is being built, the Phase progress bar shows the number of states of the mapping MDD built so far, the total number of states of the mapping MDD, and the fraction of the phase that has been completed. The Overall progress bar is also updated accordingly.
11.3.6 Command line options

The symbolic state space generator is usually run from the GUI. However, since the models are compiled and linked with the generator libraries to form an executable, the generator can also be run from the command line (e.g., from a system shell/terminal window). The executable is found in the directory

\[\text{<M"obius Project Root>/Project Name>/Solver/Solver Name>/}\]

and is named \(<\text{Solver Name}>\text{SymGen.<Arch>[_debug]},\text{ where Arch is the system architecture (Windows, Linux, or Solaris) and } _\text{debug} \text{ is appended if you build the symbolic state space generator in normal mode. The command line options and their arguments are as follows:}\]

- **-P** experimentfilepath
  Same as the flat state space generator. For more details, refer to Section 11.2.3.

- **-N** experimentname
  Same as the flat state space generator. For more details, refer to Section 11.2.3.

- **-B** experimentnumber
  Generate the state space for the experiment numbered experimentnumber, with the first experiment having number 0. For example, use “-B1” to run the second experiment (e.g., Experiment_2). Defaults to 0.

- **-c**
  Enables the compositional lumping algorithm, which is disabled by default.

- **-e, -s** db_entity, db_macserserver
  These are internal options for interaction with the database. You can safely ignore them.

- **-l** trace_level[0-1]
  Use \(trace\_level\) as the trace level described above. Only applies when you are running the debug executable.

- **-t** tracefilename
  Create a trace file named tracefilename as output. If this option is not specified, output is directed to the standard output stream.
11.3. SYMBOLIC STATE SPACE GENERATOR

-d MDD_Utable_size, union_cache_size, local_exploration_cache_size, MxD_Utable_size, submat_cache_size, add_cache_size

These are the different parameters for the symbolic data structures. They should follow the -d option in the form of a comma-separated list of non-negative numbers. The numbers in parentheses are the default value and the minimum value enforced by the symbolic SSG.

- *MDD_Utable_size* The size of the unique table of the MDD (50,000 and 1,000)
- *union_cache_size* The cache size of the union operation of the MDD (10,000 and 500)
- *local_exploration_cache_size* The cache size of the local state space exploration operation of the MDD (10,000 and 500)
- *MxD_Utable_size* The size of the unique table of the MD (10,000 and 500)
- *submat_cache_size* The cache size of the submatrix operation of the MD (5,000 and 100)
- *add_cache_size* The cache size of the add operation of the MD (5,000 and 100)

This command line option is optional. If it is not given, the default values will be used. If the option is present and the value for a parameter is zero or not given (i.e., there is no character between two consecutive commas) the default value for that parameter is used.

**Example:**

```
sssgSymGen_Linux_debug -E -B0 -l1 -ttrace.txt -d 50000,,10000,0,5000,5000
```

- `<Solver Name>` is sssg
- `<Arch>` is Linux

This command will symbolically generate the state space for the first experiment (Experiment_1) with trace level 1, place the experiment files in the current directory, and create a trace file named `trace.txt`. All parameters for the symbolic data structures except `union_cache_size` and `MxD_Utable_size` are given by the user via the -d option. For those two parameters the default values are used.
Chapter 12

Simulator

The simulator can be used to solve models using discrete event simulation. It takes the model and parameters specified by a study, links the libraries for those models together with the simulator library, and runs the executable. The simulator can be used to solve any model specified in Möbius, and can be used to solve for transient and steady state results.

12.1 Simulation Parameters

The Simulation Parameters tab on the Simulator editor allows you to specify all of the simulation parameters and is shown in Figure 12.1. The parameters on the page can be roughly broken down into four categories that proceed down the tab.

1. Study and experiment selection
2. Simulation execution parameters
3. Compilation options
4. Check boxes

12.1.1 Study and experiment selection

The top of the tab starts with the Current Study field, which contains the name of the study to be solved. The study is initially set when you create the simulation, and you are prompted to specify the solver child. You can change the study by typing a new study name into the field, or by clicking the Browse button. If you click on
Figure 12.1: Simulation Parameters tab of the Simulator.
the Browse button, the Select Solver Child window will pop up. It shows all of the studies and prompts you to select one.

The Experiment Activator button opens a pop-up window (shown in Figure 12.2) to allow you to view and select the experiments in the study. Recall that all global variable values are assigned in the study, and that each unique set of assignments is called an experiment. The pop-up window shows all of the experiments in the study, the global variable values for each experiment, and the experiments that have been activated in this simulation so that they will be solved. You can activate or deactivate an experiment by checking the box above the experiment. You can activate or deactivate all experiments by clicking on the Activate All or Deactivate All buttons.

![Experiment Activator](image)

Figure 12.2: Window for enabling and displaying experiment values.

### 12.1.2 Simulation execution parameters

The next set of parameters affect the execution of the simulation.

- The radio boxes under Simulation Type specify whether the model solution should be a steady state solution or a terminating solution. If the model contains transient rewards, the Terminating Simulation option will be enabled, and if the model contains steady state rewards, the Steady State Simulation option will be enabled. If all the rewards are either transient or steady state, only one option will be enabled, and it will be selected by default.

- The pull-down menu Random Number Generator is used to select the pseudo-random number generator to use in the solution. The default op-
tion is the Lagged Fibonacci random number generator, but the Tauseworthe random number generator can be used.

- The Random Number Seed field specifies the seed for the pseudo-random number generator. The default value is 31,415, but any value can be used. Once the seed has been specified, the sequence of numbers is deterministic. Multiple simulations with the same random number seed should generate the same results.

- The Maximum Batches field specifies the maximum number of batches (in steady state simulation) or replicas (in transient simulation) that the simulator will run. The simulator will run several batches or replicas to generate data for the confidence intervals. The simulator will continue to run more batches or replicas until all of the results have converged to their specified confidence intervals. This option will terminate the simulation if the simulator has run the maximum number of batches, even if not all of the reward variables have converged to their specified confidence intervals.

- The Minimum Batches field specifies the minimum number of batches (in steady state simulation) or replicas (in transient simulation) that the simulator will run. The simulator will run several batches or replicas to generate data for the confidence intervals. The simulator will continue to run more batches or replicas until all of the results have converged to their specified confidence intervals. This option ensures that the simulator will run at least the minimum number of batches or replicas before finishing. This is useful in simulating rare events, for which the model’s reward variables may appear to converge too early if no rare events have occurred.

- The Number of Batches per Data update field specifies how many batches or replicas the simulation executables should complete before sending those results back to the editor. If the simulation is run on multiple machines, each machine will send an update after simulating the specified number of batches or replicas.

- The Number of Batches per Display update field specifies how many batches or replicas should be received by the editor before it updates the user interface. A low number will provide the most up-to-date data, but can cause the user interface to use too many resources. A larger number will reduce

---

1The simulator is multi-threaded and can be distributed. Due to timing variations in a distributed environment, the results will not necessarily be identical when an experiment is simulated multiple times with the same random number seed, even though the random number sequences will be identical.
the load created by the user interface, but will also provide fewer updates. In
general, this number should always be greater than or equal to the number
specified in Number of Batches per update.

12.1.3 Compilation options

The next set of options control how the simulation is compiled and run, allowing
for optimizations, trace output, and specification of the run name.

- The Build Type pull-down menu allows you to set the model to run in either
  optimized or normal mode. In normal mode, the model is compiled without
  compiler optimizations, and it is possible to specify various levels of output
  by changing the trace level. In optimized mode, the model is compiled
  with compiler optimizations, and all trace output is disabled for maximum
  performance. You would normally use normal mode for testing a model and
  making sure it is behaving properly, while you would use optimized mode
  for production simulations.

- The Build Architecture pull-down menu allows you to set the model to ex-
  ecute in either 32-bit or 64-bit mode. On 32-bit machines (or machines run-
  ning a 32-bit operating system), this drop down menu will be disabled and
  only 32-bit execution is available.

- The Trace Level pull-down menu allows you to set the level of detail in-
  cluded in a trace of the simulation. The trace includes key information about
  the evolution of the simulation in a text file, which can be used to debug a
  model. The available trace levels are:

  - 0: No Output: No trace file is generated.
  - 1: Firing Time and Activity Name: This option will include the sim-
    ulation time of every activity completion, and the name of the activity
    that fired.
  - 2: Level 1 plus Minimal State: This level adds some relevant state
    information from the time of the activity firing to the previous level’s
    activity firing times and names. Specifically, it shows the values of state
    variables that are either part of the enabling conditions for the activity,
    or part of the state affected by the activity.
  - 3: Level 1 plus All State: This level prints out all of the state informa-
    tion at every activity firing, in addition to the name of the activity
    and the time it fires.
– 4: All: This level prints out all of the information from the previous levels plus more data on which activities are enabled, become enabled, or become disabled when an activity fires.

The trace-level options can generate useful information that you can use to understand the behavior of your models. However, it can also generate more information that you want. Generally, trace output is only used with a small number of batches.

• The Run name field specifies the name for this run of the simulation. This is used to name trace files and output files. Change the run name anytime you want to save existing results from a previous run of the simulator. Otherwise, the new results will overwrite any existing results.

12.1.4 Check boxes

The last set of options are set using check boxes. Most of them control the output of the simulator, while one of them controls the variance calculation.

• Store simulator console output to file stores all of the output from the editor display to a file.

• The Store observations to ASCII .csv file option can be used to save the raw simulation results to a file for later use. Again, this option is useful if you are running remote simulations over a low-bandwidth connection, because you can access the observations at a later time to get the results of the simulation. The option stores the observations in an ASCII file format. ASCII is human-readable, and can also be loaded into tools like Excel for custom analysis.

• The Store observations to binary .dat file option can be used to save the raw simulation results to a file for later use. This option is useful if you are running remote simulations over a low-bandwidth connection. You can access the observations at a later time to get the results of the simulation. The option stores the observations in a binary file, which should be smaller than the ASCII file the other option would have produced. The binary files are used by Möbius for off-line processing (see Section 12.3.2).

There used to be a checkbox in the Simulation Parameters tab called Use Jackknife Variance Calculation that made the simulator use the advanced jackknife method of variance calculation. In Möbius 1.8.0 and later, the method is always used; therefore, the checkbox has been removed.
12.2 Network Setup

Use the Network Setup tab, shown in Figure 12.3, to select the computers on which the simulation will be run. You can parallelize simulations by executing multiple replicas of the simulation on different machines at the same time, with near linear speed-up in solution time. On the left of the tab is the Available Systems list of machines or groups of machines that can be used in the solution of the model. You can select one or more machines for use by selecting their names in the list, and then clicking on the >> button. Clicking on the button will transfer all of the selected machines to the Selected Systems list. You can deselect machines in the same manner, by selecting their names in the Selected Systems list and clicking on the << button.

You can edit the list of available systems by clicking on the Edit Machine/Group Info button. Clicking on the button opens the Network Machine and Group Settings dialog, which is described in Section ??.

The Number of Processors per Experiment drop-down menu allows you to select the number of processors to use per experiment. The simulator can assign multiple machines to simulate the same experiment, and each experiment will be assigned a number of machines no greater than the specified number. Machines will be assigned to the lowest-numbered experiment that has not yet been solved and that has fewer than the specified number of machines.

The Maximize Processor Usage checkbox overrides the Number of Pro-
cessors per Experiment setting. If the Maximize Processor Usage option is enabled, the simulator will distribute experiments to all the machines, such that each experiment is running on a similar number of machines. When an experiment finishes, the machines that were assigned to it are then reassigned to the other experiments. In that way, all of the experiments are solved as quickly as possible.

### 12.3 Running Simulation

The **Run Simulation** tab, shown in Figure 12.4, is used to start and stop the simulator. As the simulation is being compiled and run, messages are produced and are shown in the **Simulation Status** area. The information includes the name of all the files being compiled and linked, and indicates when the execution of the simulation begins.

![Figure 12.4: Run Simulation tab of the Simulator.](image)

On the bottom of the tab are a couple of check boxes and a series of buttons to control the running of the simulation. The checkboxes **Skip compile** and **Force compile** control whether the underlying files are compiled before starting the simulations. By default, the simulator editor will recompile the minimum set of files before starting the simulations for the first time if neither checkbox is checked.

There are two modes for generating simulator results. The first mode is **online**, in which the simulation executables are run and the results are computed and displayed by the editor as the solution converges. The second mode is **offline**. There are two steps in off-line simulation: the first step is the launch of the
12.3. RUNNING SIMULATION

Simulation executables with the binary output file option enabled, and the second step is the processing of the binary output files when the simulator processes have completed. The two modes are discussed in the following sections.

12.3.1 On-line simulation

The Start Simulation button starts the simulator. Any files that should be recompiled are recompiled, and experiments are assigned to the selected machines.

The Stop Simulation button will stop a running simulation. It will stop all of the simulations started on remote machines, and finish any remaining result computation.

The Client Information dialog, shown in Figure 12.5 is used to check the status of a client machine during solution. The Client Information window will pop up with a list of client machines used in this simulation. When you double-click on any of the system names, the Client Status window, shown in Figure 12.6 will pop up to provide the name of the system, the architectures, and the operating system.

Figure 12.5: Pop-up window to check the status of the client machines.

Figure 12.6: Pop-up window that displays the status of a client machine.
12.3.2 Off-line simulation

The Process Binary File button allows you to process a binary output file that was generated if you checked the Store observations to binary .dat file option, as described in Section 12.1.4. When you press the button, a file browser will open to allow you to select the appropriate binary data output file(s). Once you have selected the files, Möbius reads them and processes the data as if the simulation was running on-line, and displays the results after all data has been read.

Binary data files can be generated through multiple simulation runs. You can launch runs with the Möbius simulation editor or launch the simulation executable directly via the command line.

One reason to use the off-line processing capability is to generate solutions without having to run the Möbius graphical interface. You can launch jobs on the command line (directly or via user-constructed command line scripts), collect the results, and transfer them to a machine with graphical access on which the data files can be processed to compute the results.

Another reason to use off-line processing is to generate more simulation replications than could be generated in one session. Repeated executions of the simulator (using a unique random seed and run name each time) will produce multiple sets of data files. These files can be processed together at one time to generate results based on all the runs.

12.4 Simulation Info

The Simulation Info tab displays the results for the experiments as they become available, and is shown in Figure 12.7. At the top left of the tab is the list of all the enabled experiments and their statuses. The Status column indicates whether the experiment has not started, is running, or has completed. The # CPUs column indicates how many processors are currently assigned to the given experiment, and the Batches column indicates how many batches or replicas of that experiment have completed and been reported back to the user interface. You can select an experiment by clicking on its row in the table. Selecting the experiment in the list updates the Selected Experiment display in the lower half of the window. The Selected Experiment display shows the simulation progress by reporting the number of replications computed and showing the mean and variance values and confidence intervals for each reward variable. When the result converges so that the computed confidence interval is within the tolerances specified in the Simulation panel of the Performance Variable Editor (see Section 7.1.6), the color of the reward variable row changes from red to blue.

There are three buttons to the right of the experiment list:
Figure 12.7: Simulation Info tab of the Simulator.
• The Terminate Selected Experiment button stops the solution of the currently selected experiment. All machines that are simulating that experiment stop doing so, and are reassigned.

• The Terminate All Experiments button stops the solution of all the experiments. The simulator receives the final updates and displays the final results.

• The Show Results button opens the Simulation Results window, which is discussed in the next section.

### 12.5 Simulation Results

The results of the simulation are displayed in the Simulation Results tab when the simulation finishes or the user clicks on the Show Results button of the Simulation Info tab. The window displays the human-readable text output created by the simulator. The file includes all of the parameters for the model, all of the experiments, and all of the results. In addition to the human-readable output, a comma-separated-value file (.csv) is also created. The csv file is designed to be simple to import into third-party plotting and analysis tools (such as Excel).

### 12.6 Command Line Options

The simulator is normally run from the simulator editor. However, it can also be run from the command line. The simulator executable resides in the project tree, in a path following this pattern: `<Möbius Project Root>/<Project Name>/Solver/<Solver Name>/`. It can be run with the command line options listed in Table 12.1. These options will be shown if you execute the simulator on the command line, without any options.
12.6. COMMAND LINE OPTIONS

Table 12.1: Command line options for the Möbius simulator.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
</table>
| -a <name> | write ASCII output to file `<name>`.
| -b <name> | write binary output to file `<name>`.
| -c <num> | Client identification number (default = 0).
| -e <num> | Experiment to run, “Experiment 1” is num=0 (default = 0).
| -f <name> | Trace file name (default=stdout).
| -l <num> | Trace level (0-off, 1,2,3,4 [default = 0]).
| -n <num> | Num observations sent in 1 update to simulator editor (default = 1000).
| -N <num> | Maximum number of batches to run (default = infinite).
| -p <num> | Communication port of network server (default = 10000).
| -r <num> | Random number generator (0 = lagged Fibonacci [default], 1 = Tauseworthe).
| -s <num> | Random number seed (31,415 [default]).
| -t <num> | Simulation type (1 = terminating [default], 0 = steady state).
| -w | Simulator will wait to send output until the socket connection has been established. Used when the simulator is launched by the Möbius editor.
Chapter 13

Numerical Solvers

This chapter discusses the numerical (or analytic) solvers that are supported in Möbius. A number of solvers are available for computing solutions to various types of performability measures. This chapter begins by explaining how a new solver can be created. It then gives guidelines for choosing the appropriate solver for a given model class and measures of interest. Finally, it discusses in detail all of the available solvers and their relevant parameters. The discussion of each solver includes tips on pitfalls to avoid and hints for effective use of the solver.

13.1 Creating a New Numerical Solver

All of the solvers presented in this chapter compute their solutions by analyzing the underlying state space of a model. The state space is generated using the state space generator, as described in Chapter 11.3. You can create a new numerical solver by selecting Numerical on the Project panel and clicking the button New. A dialog window will appear, requesting a name and the type of solver to use. After you have entered the name and chosen the type of solver, a dialog appears, requesting the name of the state space to be analyzed. You can choose the desired state space by clicking on one of the choices presented in the dialog window Select Numerical Child. After you select the state space, the main editor for the numerical solver appears. You use the editor to enter the relevant parameters for your solver. The parameters corresponding to each solver are explained in the sections below. After a solver has been created, you can open it for revision by simply double-clicking on its name or by selecting its name and clicking the Open button on the Project panel.
13.2 Choosing the Appropriate Numerical Solver

The choice of which numerical solver to use depends on the type of a solvable model and the type of performability measures of interest. The type of a solvable model is determined by the distributions of its actions (see Section 1.2.5 for an explanation of the numerous distributions supported by Möbius). A solvable model may be one of two types:

**Markov Model:** All timed actions in models of this type are exponentially distributed.

**Deterministic Model:** Timed actions in models of this type may be a mixture of exponential and deterministic distributions. However, there can be no more than one enabled deterministic action at any time, and the time of the deterministic activity cannot depend on the marking of the model.

A performability measure has three defining attributes. These attributes, together with the model type, determine the appropriate solver to use. They are as followings:

- Whether the performability measure is obtained in transient or steady state;
- Whether the performability measure is measured at an instant of time or over an interval of time;
- Whether the mean, variance, or both are desired.

The selection of an appropriate solver based on these attributes and model type is summarized in Table 13.1. In addition, a list of advantages and disadvantages in using the numerical solvers is provided below. You should consider these issues in order to make effective use of the solvers.

**Advantages of analytic solution**

- Exact computation of solution is carried out, as opposed to simulation, in which the solution is estimated along with a confidence interval indicating how likely it is that the estimated solution is the solution of the exact computation.

- For the instant-of-time performability variables, distributions can be obtained without extra cost beyond that of obtaining their mean and variance.

- Accuracy of the solution can, for most solvers, be increased without excessive increase in the computation time (within the limitations stemming from machine accuracy).

Advantages of analytic solution
## 13.2. Choosing the Appropriate Numerical Solver

Table 13.1: Models and measures versus the applicable numerical solvers.

<table>
<thead>
<tr>
<th>Model Class</th>
<th>Steady-state or Transient</th>
<th>Instant-of-time or Interval-of-time</th>
<th>Mean, Variance, or Both</th>
<th>Applicable Analytic Solver</th>
</tr>
</thead>
<tbody>
<tr>
<td>All actions exponential</td>
<td>Steady-state</td>
<td>Instant-of-time</td>
<td>Both</td>
<td>(dss), (iss) and (tss)</td>
</tr>
<tr>
<td></td>
<td>Transient</td>
<td>Instant-of-time</td>
<td>Both</td>
<td>(trs) and (atrs)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Interval-of-time</td>
<td>Mean</td>
<td>(ars)</td>
</tr>
<tr>
<td>Exponential and deterministic actions</td>
<td>Steady-state</td>
<td>Instant-of-time</td>
<td>Both</td>
<td>(dis) and (adiss)</td>
</tr>
</tbody>
</table>

\(a\) if only rate rewards are used, the time-averaged interval-of-time steady-state measure is identical to the instant-of-time steady-state measure (if both exist).

\(b\) \(dss \equiv \) Direct Steady-State Solver

\(c\) \(iss \equiv \) Iterative Steady-State Solver

\(d\) \(tss \equiv \) Takahashi Steady-State Solver

\(e\) \(trs \equiv \) Transient Solver

\(f\) \(atrs \equiv \) Adaptive Transient Solver

\(g\) \(ars \equiv \) Accumulated Reward Solver

\(h\) provided that the instant-of-time steady-state distribution is well-defined. Otherwise, the time-averaged interval-of-time steady-state variable is computed, and only results for rate rewards should be derived.

\(i\) \(diss \equiv \) Deterministic Iterative Steady-State Solver

\(j\) \(adiss \equiv \) Advanced Deterministic Iterative Steady-State Solver
Disadvantages of analytic solution

- Analytic solvers are not available for all models. The models must belong to one of the two model classes discussed above.

- The state space size of the generated model must be finite. Moreover, it cannot be too large relative to the memory of the computer being used. The iterative solvers iss, trs, and ars can usually deal with models having several million states. The other solvers demand additional memory on top of that needed for the storage of the transition matrix. See the discussion in the sections specific to the different solvers for more details.

- It is a challenge to create models from which all the desired performability results can be derived, but which have a state space small enough to allow for analytic solution. Considerable payoffs can be expected from exploring state space reduction approaches. In that respect, the use of the Rep construct in the composed model can be very helpful.

- Analytic solution is time-consuming if one is dealing with stiff models. A prominent class of stiff models is the one with a wide range of expected action completion times. An example is a dependability model in which there are long periods until component failures and relatively fast repairs.

Additional constraints

In addition to the constraints already imposed on the two model classes discussed above, analytic solvers of either class cannot be used to solve PEPA models which contain guard conditions.

13.3 Detailed Descriptions of Supported Numerical Solvers

The technical details of the supported numerical solvers are described below. Details are given about how their required parameters are entered into Möbius and how their output should be interpreted. The discussion includes hints on pitfalls to avoid and ways to make effective use of the solvers.

13.3.1 Common options among all analytic solvers

The following options are common to all the solvers:

- **Accuracy** is an integer indicating the degree of accuracy that the user wishes in terms of the number of digits to the right of the decimal point. Usually, the
13.3. **DETAILED DESCRIPTIONS OF SUPPORTED NUMERICAL SOLVERS**

default is nine digits. If the number of digits specified exceeds the machine accuracy, then the machine accuracy is used. Note that the precise meaning of the accuracy setting depends on the solver. The precise meaning of **Accuracy** will be stated in the below discussion of the individual solvers.

- The **State Space Name** specifies the generated state space to use for computing numerical solutions. You may choose among the available state spaces by pressing the button “...” to the right of the text field next to **State Space Name**.

- If an **Output File Name** is given as (for example) “outfile”, the results are written into the file `outfile_Experiment_i.txt`, where `i` is the experiment number. This file is put in the solver’s results directory, `MobiusProject/projectname/Result/solvername`. If no output file is given, the output goes to the control panel window.

- If a **Debug File Name** is given as (for example) “debugfile”, debug information is written into the file `debugfile_Experiment_i_debug.txt`, where `i` is the experiment number. This file is put in the solver’s results directory, `MobiusProject/projectname/Result/solvername`. If no debug file is given, no debug information is generated. The debug information consists essentially of detailed information regarding the solver. It can be useful for determining whether a solution converges, but usually the same information can be obtained more naturally by setting the **Verbosity**. The **Verbosity** option will be discussed with respect to each individual solver.

- **Plot Complementary Distribution** is not currently supported by Möbius.

- If **Run in the Background** is selected, the solver process is run in the background so that other control panel options can still be used while the solver is running. If no output file is specified, output automatically goes to the control panel window.

The output file of each solver contains various information. It first itemizes the options that were used, including defaults, and it contains the results of the solution process. It also records the following information:

- The **project**, **study**, and **experiment** for which the results were computed.

- The **Global variable settings**, which are the values assigned to all the global variables in the chosen experiment.
CHAPTER 13. NUMERICAL SOLVERS

- The *Number of states in process*, which is the number of states that were generated by the state-space generator.

- The *Number of non-zero elements*, which is the number of non-zero elements in the transition matrix.

- The *Computation Time*, which is the total execution time, equal to the sum of user time and system time.

- The *User Time*, which is the total amount of time spent executing in user mode.

- The *System Time*, which is the total amount of time spent executing in system mode.

13.3.2 Direct steady-state solver

The direct steady-state solver (dss) solves for instant-of-time variables with $t \to \infty$, using a numerically stable version of LU decomposition [37]. It does so by solving the system of equations given by $\pi Q = 0$, with the additional constraint that the sum of all elements in the vector $\pi$ must equal one. $\pi$ is the state occupancy probability vector, and $Q$ is the generator matrix. The generator matrix is obtained from the SAN through use of the state space generator (see Chapter 11.2). The solver uses two methods to reduce the fill-ins (non-zero elements) of the matrix during solution: (1) the improved generalized Markowitz strategy, which selects a next pivot element based on a heuristic that can reduce fill-in, and (2) a technique that sets elements below some threshold (which is tunable, see the options below) to zero during the execution of the solution algorithm [37]. If the problem is not too large, the solver then uses iterative refinement to reach a correct final solution. The solver calculates the mean and variance for each performability variable. The means and variances are recorded in textual form in the output file. Figure 13.1 shows the editor for the direct steady-state solver and its available options and adjustable parameters. The options are as follows:

- The **Rows** is an integer representing the number of rows to search for a pivot. Zero is the default, meaning pivoting is turned off by default. A value of 2 or 3 is recommended [37].

- The **Stability** is a short integer representing the “grace” factor by which elements may become candidates for pivots. 0 is the default, meaning pivoting is turned off. A stability factor between 4 and 16 is recommended in the literature; see [37], for example.
13.3. **DETAILED DESCRIPTIONS OF SUPPORTED NUMERICAL SOLVERS**

Figure 13.1: The Direct Steady-State Solver dss editor and available options and parameters.

- The **Tolerance** is a double that, when multiplied by the smallest matrix element, is the threshold at which elements will be dropped in the LU decomposition. 0.0 is the default, which implies that no dropping takes place. In general, it is recommended [37] that the drop tolerance be chosen to be two to five orders of magnitude smaller than the smallest matrix element, i.e., choose the **Tolerance** between $10^{-2}$ and $10^{-5}$.

- **Verbosity** ($n$) sets the trace level for intermediate output. The default is no intermediate output. If $n > 0$, then the message “completed column number n” is printed after every $n$ iterations during the LU decomposition computation, forward substitution, backward substitution, and iterative refinement.

The output file contains the means and variances of the performability variables. It also records the following information:

- whether iterative refinement was used,
- the drop tolerance,
- the number of non-zero elements in the original matrix,
- the number of non-zero elements in the factorized matrix,
- if iterative refinement was used, the maximum difference between the cells in the $\pi Q$ and zero vectors; if iterative refinement was not used, the relative error,
• the number of correct decimal digits in the state probabilities,
• the number of zeros in the factorized matrix,
• the number of elements dropped,
• the number of new pivots selected.

Pitfalls and Hints

• dss can be used if the steady-state distribution of the Markov model consists of a single class of recurrent non-null states. For instance, dss cannot be applied to a model with multiple absorbing states. If it is, the message “invnmnorm: zero diagonal element” will appear and the performance variable will take the NaN (Not a Number) value. To find out whether the model has absorbing states, apply the Flag Absorbing States option in the state space generator.

• dss is effective when relatively small models are considered, because in the process of computing the steady-state probabilities, the original transition matrix is transformed into a matrix with many non-zero elements. Sparse matrix methods, which use the fact that elements equal to 0 do not have to be stored, can then no longer be profitably applied. This is known as the fill-in problem. Especially when large models are considered, fill-ins can become a serious bottleneck, because the order of non-zero elements is, in general, quadratic in the size of the state space. Consequently, dss cannot be used to solve models having state spaces that are larger than several thousand states. Note that the setting of the drop tolerance may be used to partially overcome fill-ins of the matrix.

• The CPU time required by dss also increases in the number of states (with a power of 3). The iterative solver iss often becomes faster than the direct solver dss when the state space size increases.

13.3.3 Iterative steady-state solver

The iterative steady-state solver (iss) solves for instant-of-time variables with $t \to \infty$. Two iterative methods are provided in Möbius for this solver: successive over-relaxation (SOR) and Jacobi. iss solves the system of equations given by $\pi Q = 0$, where $\pi$ is the state occupancy probability vector and $Q$ is the generator matrix. The algorithm guesses $\pi$, calculates $\pi Q$, and then comes up with a new guess related to the difference between the answer and the zero vector. It continues until
13.3. DETAILED DESCRIPTIONS OF SUPPORTED NUMERICAL SOLVERS

The maximum difference between the cells in the two vectors is within the error bounds.

The initial guess for $\pi$ is equal probabilities for all states. The acceleration factor to be used must be selected by the user. Because of its more modest space requirements, this solver can be applied to larger models than dss can, but it is not guaranteed to converge for all state spaces and initial conditions. It calculates the mean and variance for each performability variable. The means and variances are recorded in textual format in an output file. Figure 13.2 shows the editor for the iterative steady-state solver and its available options and adjustable parameters. The options are as follows:

- **Type** of iterative method refers to either SOR or Jacobi. In general, SOR should be used because it converges in fewer iterations and it has smaller memory requirements than Jacobi. However, because SOR requires that the underlying generator matrix be accessible in column order, it cannot be used with state spaces generated by the symbolic state-space generator (see Chapter 11.3 for more details). Use Jacobi instead for these state spaces because accesses to the matrix elements are not provided in any particular order.

- **Weight** is a double representing the acceleration factor. 1.0 is the default. A value of 1.0 reduces the SOR method to Gauss-Seidel. Values be-

Figure 13.2: **Iterative Steady-State Solver** is editor and available options and parameters.
tween 1.0 and 2.0 may accelerate convergence. Values between 0.0 and 1.0 are less subject to divergence.

- The Max Iterations is an integer representing the maximum number of iterations that will be performed before the solver terminates. 300,000 is the default.

- The Verbosity \( n \) sets the trace level of intermediate output. The default is no intermediate output. If \( n > 0 \), then the accuracy is printed after every \( n \) iterations.

The output file contains the mean and variance of the performability variables. It also contains the following information:

- the number of iterations required for convergence,
- the maximum difference, which is the maximum difference (over all the states) between the solution in the last two iterations.

**Pitfalls and Hints**

- The iss solver can be used for many practical models. A required condition for these models is that they are ergodic Markovian models. Otherwise, if the model contains one or more absorbing states, iss cannot be used; it will produce the message `iss_solver: zero on the diagonal` and quit. To find out whether the model has absorbing states, apply the Flag Absorbing States option in the state space generator.

- The iss algorithm stops when the largest difference of the state probabilities between two iterations (at that moment not yet normalized to sum to 1) is less than the specified error. This stopping criterion does not directly relate to the error between the derived and the real state probabilities, let alone between the derived and the real performability variables. A value of \( 10^{-9} \) for the Accuracy will usually be sufficient.

- As a rule of thumb, the additional time it takes to get \( n \) times as accurate a result is of the order \( \log_{10} n \). Hence, increasing accuracy tends to be not too costly. Of course, the machine accuracy can never be exceeded.

- First try iss with Weight equal to 1. This usually leads to quick solutions. A higher weight may decrease the number of iterations; however, an (even slightly) too high weight can dramatically increase the necessary number of iterations. If iss does not converge for a Weight of 1.0, try values lower than
13.3. DETAILED DESCRIPTIONS OF SUPPORTED NUMERICAL SOLVERS

1.0. Typically, making Weight < 1 improves convergence, while Weight > 1 decreases the number of iterations if convergence has already been assured. Note that the value of Weight should be between 0 and 2.

- The iss solver usually derives results in a reasonable amount of time. If the state space is large, more computation is necessary per iteration, but the number of iterations is often relatively low. Therefore, initially use the default for the number of iterations. If iss does not converge within a reasonable number of iterations, you may have set the accuracy too high for the machine being used. Be careful in choosing an accuracy smaller than $10^{-10}$ (i.e., an input value of 10 for Accuracy). You can check the progress toward convergence of iss by using the Verbosity option.

- Some models may require many iterations. They are called stiff models and belong to the class of nearly-decomposable models. They occur, for example, when the performance of a system quickly reaches some state that initially seems to be a steady state for any system configuration; however, the actual steady state is a different state and changes from the initial state take place very slowly and infrequently. This is due to the large difference in the rates of the actions in the individual models.

13.3.4 Takahashi steady-state solver

The Takahashi steady-state solver (tss) solves for instant-of-time variables with $t \to \infty$. It is an iterative solver meaning that it initially guesses an initial value for the $\pi$ vector (state occupancy probability), and then at each iteration, it computes a new value for $\pi$ that may be “closer” to the value that satisfies $\pi Q = 0$. For the Takahashi solver, the initial guess for $\pi$ is equal probabilities for all states.

The solver is an instance of a general class of iterative solvers called IAD (Iterative Aggregation/Disaggregation) solvers [35]. In IAD solvers, the $Q$ matrix is partitioned into $N \times N$ blocks. As their names imply, each (main) iteration of the solution algorithm of IAD solvers is comprised of two phases: 1) aggregation and 2) disaggregation.

In the first phase, an $N \times N$ aggregation matrix $A$ is constructed by computing each of its elements from the corresponding block of $Q$. Then, the system of equations $\xi A = \xi$ is solved using the SOR iterative steady-state solver (see Section 13.3.3) to compute the vector $\xi$. In the second phase, $N$ systems of equations each of which derived from a column of $N$ blocks of $Q$ are solved by the SOR solver. Based on the solution vectors computed in the two phases, a new value for $\pi$ is computed. The iterations are repeated until either the maximum difference between the elements of the previous $\pi$ and the new $\pi$ is less than a given error.
boundary (given by the Accuracy parameter described below) or the number of iterations exceeds a given maximum value (given by the Max Iterations parameter described below).

Because of its more modest space requirements, this solver can be applied to larger models than dss can, but it is not guaranteed to converge for all state spaces and initial conditions. It calculates the mean and variance for each performance variable. The means and variances are recorded in textual format in an output file. Figure 13.3 shows the editor for the iterative steady-state solver and its available options and adjustable parameters. The options are as follows:

- The Number of Divisions is the positive integer $N$ mentioned above. Default value is 5.
13.3. DETAILED DESCRIPTIONS OF SUPPORTED NUMERICAL SOLVERS

- The **Accuracy** determines the accuracy used for checking convergence condition on $\pi$. Default value is 9. See Section 13.3.1.

- The **Max Iterations** is an integer representing the maximum number of main iterations that will be performed before the solver terminates. Default value is 300,000.

- The **Aggregation Phase Accuracy** is the accuracy of the iterative SOR solver of the aggregation phase. Default value is 9.

- The **Aggregation Phase Max Iterations** is an integer representing the maximum number of iterations (for each main iteration of the tss solver) that the SOR solver of the aggregation phase performs before it terminates. Default value is 300,000.

- The **Aggregation Phase Weight** is a double representing the acceleration factor for the SOR solution of the aggregation phase. 0.95 is the default. See Section 13.3.3 for more information.

- The **Disaggregation Phase Accuracy** is the same as **Aggregation Phase Accuracy** except that it is used in disaggregation phase of each main iteration.

- The **Disaggregation Phase Max Iterations** is the same as **Aggregation Phase Max Iterations** except that it is used in disaggregation phase of each main iteration.

- The **Disaggregation Phase Weight** is the same as **Aggregation Phase Weight** except that it is used in disaggregation phase of each main iteration.

- The **Verbosity** ($n$) sets the trace level of intermediate output. The default is no intermediate output. If $n > 0$, then some details about the progress of the solution algorithm is printed out. $n >= 2$ is more verbose than $n = 1$.

**Pitfalls and Hints**

- This solver is most suitable for NCD (Nearly Completely Decomposable) Markov chains [35]. An NCD Markov chain is one whose state space can be partitioned into disjoint subsets such that the transition rates among states of the same subset are relatively high and among states of different subsets are relatively low.
• In the current version of Möbius, the Takahashi solver is applicable to state spaces generated from the flat state-space generator but not to ones generated from the symbolic state-space generator. The reason is that a generator matrix can be divided into blocks in a straightforward manner when it is represented as a sparse matrix but that it is not the case when it is represented symbolically.

• Currently, Möbius does not apply any particular automatic method to compute the optimum value for $N$, to compute the optimum size of the blocks, or to determine the best reordering of the states. The reason is that optimizing each of those parameters is, in some cases, more difficult than the solution of the Markov chain itself, if not intractable.

### 13.3.5 Deterministic iterative steady-state solver

The deterministic iterative steady-state solver (diss) solves for instant-of-time variables with $t \to \infty$ using uniformization and successive over-relaxation (SOR) \[34\]. diss should be used for the steady-state solution when there is at least one deterministic action in the model. Solution is restricted to models in which there is no more than one deterministic action enabled in each process state. The state-space generator can be used to detect states in which more than one deterministic action is enabled. The solution algorithm is similar to that used by iss, but uniformization is used to compensate for the deterministic action. You must select the acceleration factor. diss calculates the mean and variance of each performability variable. The means and variances are given in textual format in an output file. Figure \[13.4\] shows the editor for the deterministic iterative steady-state solver and its available options and adjustable parameters. The options are as follows:

• The **Weight** is a double-precision float representing the acceleration factor. 1.0 is the default. A value of 1.0 reduces the SOR method to Gauss-Seidel. Values between 1.0 and 2.0 may accelerate convergence. Values between 0.0 and 1.0 are less subject to divergence.

• The **Error Tolerance** is a short integer representing a negative power of 10 (i.e. error tolerance) for truncation of infinite series during calculation of the Poisson probabilities. 11 is the default value. Increasing the error tolerance may increase the solution time. The solution time can be reduced if more error can be tolerated.

• The **Max Iterations** is an unsigned long representing the maximum number of iterations that will be performed before the solver terminates. 300,000 is the default.
13.3. **DETAILED DESCRIPTIONS OF SUPPORTED NUMERICAL SOLVERS**

Figure 13.4: Deterministic Iterative Steady-State Solver **diss** editor and available options and parameters.

- The **Verbosity** \( (n) \) sets a trace level of intermediate output. The default is no intermediate output. If \( n > 0 \), then the accuracy is printed after every \( n \) iterations.

- If **Detect Steady State** is selected, the solver detects the steady state before it detects the right truncation point, if possible [34]. It can reduce the solution time, but you should compare the results obtained with and without this option to make sure that steady state has not been falsely detected.

- If **Save C matrix in file** is selected, the solver saves rows of the \( C \) matrix in a file instead of keeping them in memory. Saved rows are read back at appropriate times. This option should be used when the solver complains about a shortage of memory while solving a model with a large state space.

- If **Save P matrix in file** is selected, the solver saves rows of the \( P \) matrix in a file instead of keeping them in memory. Saved rows are read back at appropriate times. This option should be used when the solver complains about a shortage of memory while solving a model with a large state space.

The output file contains the means and variances of the performability variables. It also contains the following information:

- The index of deterministic action considered, which is an index into an internal data structure of the deterministic action that is being processed. The
number itself is not useful, only the fact that progress is being made.

- The left truncation point, which is the number of iterations below which uniformization does not collect results.

- The right truncation point, which is the number of iterations above which uniformization does not collect results.

- The number of iterations required for convergence.

- The maximum difference, which is the maximum difference between the cells in the $\pi Q$ and zero vectors, which represents the error. The truncation error is not reported, but is bounded by the specified error tolerance.

**Pitfalls and Hints**

- The diss solver suffers from the fill-in problem, albeit to a lesser extent than the dss solver. For every marking in which a deterministic action is enabled, the transition probabilities to all the markings that can be reached during the deterministic time are computed. Depending on the model, this results in a considerable number of fill-ins if a high percentage of the markings enable a deterministic action. One example that leads to many fill-ins is a single buffer with a deterministic server; the deterministic action is enabled in all markings that represent at least one job in the buffer. Consequently, considerable fill-ins will occur.

- The instant-of-time steady-state measure is not necessarily defined for models with deterministic actions, because periodic behavior may exist. The outcome of diss can in that case be interpreted as the time-averaged interval-of-time steady-state measure. However, that interpretation is valid when only rate rewards are considered (i.e., no measures with impulse reward are defined). Furthermore, the variance and distribution that are derived do not have any meaning for the interval-of-time variables.

- diss cannot solve for deterministic actions with delay values that are marking-dependent. If the specified model contains a marking-dependent deterministic action, the obtained results should be discarded.

**13.3.6 Advanced deterministic iterative steady-state solver**

The advanced deterministic iterative steady-state solver (adiss) solves for instant-of-time variables with $t \to \infty$, using a two-stage iterative method that takes advantage of the structure of the probability transition matrix of an embedded Markov
13.3. **DETAILED DESCRIPTIONS OF SUPPORTED NUMERICAL SOLVERS**

Figure 13.5: **Advanced Deterministic Iterative Steady-State Solver** (adiss) editor and available options and parameters.

chain. Normally, for a large Markov chain, the whole probability transition matrix, \( P \), cannot be stored in core memory and it must be stored on disk. However, that drastically increases the computation time. adiss exploits the structure of \( P \) by using a “decomposability factor,” \( \epsilon \), to decompose it into two matrices, \( P^s \) and \( P^l \), such that \( P = P^s + P^l \) and all elements of \( P^s \) are less than \( \epsilon \). For large models, \( P^s \) can be stored on disk, while \( P^l \) can be stored in core memory so that it can be accessed more often in the iterative process. Solution is restricted to models in which there is no more than one deterministic action enabled in each process state. adiss calculates the mean and variance of each performability variable. The means and variances are given in textual format in an output file. Figure [13.5](#) shows the editor for the advanced deterministic iterative steady-state solver and its available options and adjustable parameters. The options are as follows:

- **The Decomposability Factor** is a double-precision float representing the threshold for decomposing the probability transition matrix \( P \).
- **The Weight** is a double precision float representing the acceleration factor. 1.0 is the default. A value of 1.0 reduces the SOR method to Gauss-Seidel. Values between 1.0 and 2.0 may accelerate convergence. Values between 0.0 and 1.0 are less subject to divergence.
• The **Error Tolerance** is a short integer representing a negative power of 10 (i.e., error tolerance) for truncation of infinite series during calculation of the Poisson probabilities. 11 is the default value. Increasing the error tolerance may increase the solution time. The solution time can be reduced if more error can be tolerated.

• The **Max Iterations** is an unsigned long representing the maximum number of iterations that will be performed before the solver terminates. 300,000 is the default.

• The **Ps File Location** is the path to the file containing the previously stored matrix $P^s$.

• The **Verbosity** ($n$) sets a trace level of intermediate output. The default is no intermediate output. If $n > 0$, then the accuracy is printed after every $n$ iterations.

• If **Do Not Compute C or P Matrices** is selected, the $P$ and $C$ will not be computed explicitly.

• If **Store Ps Matrix in Memory** is selected, the solver stores the matrix $P^s$ in memory to help speed up the solution process. If the solver complains of insufficient memory, this option should be selected so that the matrix is stored in a file.

• If **Do Not Save C Matrix in File** is not selected, the solver saves rows of the $C$ matrix in a file instead of keeping them in memory. Saved rows are read back at appropriate times. This option should be used when there is enough main memory to store the matrix. Storing the matrix in memory will help to speed up the solution process.

• If **Do Not Save P Matrix in File** is not selected, the solver saves rows of the $P$ matrix in a file instead of keeping them in memory. Saved rows are read back at appropriate times. This option should be used when there is enough main memory to store the matrix. Storing the matrix in memory will help to speed up the solution process.

• If **Detect Steady State** is selected, the solver detects the steady state. It can reduce the solution time, but you should compare the results obtained with and without this option to make sure that steady-state is not falsely detected.

The output file contains the means and variances of the performability variables. It also contains the following information:
13.3. Detailed Descriptions of Supported Numerical Solvers

- The index of deterministic action considered, which is an index into an internal data structure of the deterministic action that is being processed. The number itself is not useful, only the fact that progress is being made.

- The left truncation point, which is the number of iterations below which uniformization does not collect results.

- The right truncation point, which is the number of iterations above which uniformization does not collect results.

- The number of iterations required for convergence.

- The maximum difference, which is the maximum difference between the cells in the $\pi Q$ and zero vectors, which represents the error. The truncation error is not reported, but is bounded by the specified error tolerance.

Pitfalls and Hints

- The adiss solver suffers from the fill-in problem, albeit to a lesser extent than the dss solver. For every marking in which a deterministic action is enabled, the transition probabilities to all the markings that can be reached during the deterministic time are computed. Depending on the model, this results in a considerable number of fill-ins if a high percentage of the markings enable deterministic actions. One example that leads to many fill-ins is a single buffer with a deterministic server; the deterministic action is enabled in all markings that represent at least one job in the buffer. Consequently, considerable fill-ins will occur.

- The instant-of-time steady-state measure is not necessarily defined for models with deterministic actions, because periodic behavior may exist. The outcome of adiss can in that case be interpreted as the time-averaged interval-of-time steady-state measure. However, that interpretation is valid when only rate rewards are considered (i.e., no measures with impulse reward are defined). Furthermore, the variance and distribution that are derived do not have any meaning for the interval-of-time variables.

- adiss cannot solve models having deterministic actions with delay values that are marking-dependent. If the specified model contains a marking-dependent deterministic action, the obtained results should be discarded.

- For decomposability factors equal to zero, the solution method of this solver is the same as that of the power method.
13.3.7 Transient solver

The transient solver (trs) solves for instant-of-time variables with $t < \infty$ using randomization (also known as uniformization). It calculates the mean and variance of each performability variable for the time points defined for the reward variable within the reward model. Uniformization is based on the idea of subordinating a Markov chain to a Poisson process. It is computationally efficient, preserves matrix sparsity, and solves to user-specified tolerances. Furthermore, computation of state probabilities in the uniformized Markov chain and computation of Poisson probabilities can both be done in a numerically stable manner. The means and variances are given in textual format in an output file. Figure 13.6 shows the editor for the transient solver and its available options and adjustable parameters. The options are as follows:

- The Verbose $(n)$ sets a trace level of intermediate output. The default is no intermediate output. If $n > 0$, then an intermediate statement is printed after computation of every $n$ columns of the power transition matrix.

The output file contains the means and variances of the performability variables. It also contains the following information:
13.3. DETAILED DESCRIPTIONS OF SUPPORTED NUMERICAL SOLVERS

- The rate of the Poisson process used to do the uniformization.
- The number of states with positive rewards.
- The number of time points.
- For each time point, the left truncation point, number of iterations and error.

Pitfalls and Hints

- The computation time of \( \text{trs} \) is determined primarily by the number of iterations. A simple way to estimate the number of iterations is to multiply the required time instant by the rate of the Poisson process. The rate of the Poisson process is equal to the highest outgoing rate over all the states of the Markov process (the outgoing rate of a state is given by the sum of all the exponential rates of transitions out of the state). As a consequence, the time complexity of the algorithm increases linearly with \( t \).

- From the previous item, it follows that \( \text{trs} \) will be more time-consuming for models with high rates of the exponential distribution relative to the time points of interest. A class of models that has that kind of stiffness can be found in reliability evaluation if repairs occur relatively fast and failures occur rarely. In such models, the rate of the Poisson process is dictated by the fast repairs, but the time points of interest are often in the scale of the times between failures. For example, for a system in which component failures occur on the average once every ten days and repairs take on the order of an hour, one’s interest will typically be in the transient behavior over relatively long periods (e.g., the probability the system is up at the end of the year).

- For large values of \( t \), the result becomes identical to the steady-state result, and will not change further if \( t \) increases. Use the \( \text{iss} \) solver to determine if this is occurring.

- At time \( t = 0 \), the SAN model is in the initial marking with probability 1. In Möbius it is not possible to specify another initial distribution. To change the state at \( t = 0 \), alter the initial marking of places in the SANs.

13.3.8 Accumulated reward solver

The accumulated reward solver (ars) solves for transient interval-of-time variables, i.e., for intervals \([t_0, t_1]\) for which both \( t_0 \) and \( t_1 \) are finite. The starting and stopping times are specified for each interval or time-averaged interval of time reward.
variable in the reward model. The ars solver gives the expected accumulated reward, as well as the expected time-averaged accumulated reward over the interval. The results are derived by uniformization. Figure 13.7 shows the editor for the transient solver and its available options and adjustable parameters. The options are similar to the transient reward solver options.

The output file contains the means, both time-averaged and accumulated, of the performability variables. It also contains additional information similar to that described for the trs solver.

Pitfalls and Hints

- The ars solver is an extension of the trs solver, and the remarks for trs apply here as well.

13.3.9 Adaptive transient solver

The adaptive transient solver (ats) solves for instant-of-time variables with $t < \infty$ using randomization (also known as uniformization) It calculates the mean and variance of each performability variable at the time points specified in the reward
13.3. **DETAILED DESCRIPTIONS OF SUPPORTED NUMERICAL SOLVERS**

The method used by ats is based on the same method described above for the transient solver, trs. However, ars is more efficient than trs for stiff models in which there are large (orders of magnitude) differences in the rates of actions. This method works by dividing the computation into time domains of slow and fast rates and adapting the uniformization rates to the time domains. Initially, “adaptive uniformization” \cite{38} is used until a “switching time.” After that time, standard uniformization is used. In effect, this method attempts to reduce the number of iterations needed to compute the solution. The means and variances are given in textual format in an output file. Figure 13.8 shows the editor for the transient solver and its available options and adjustable parameters. The options are as follows:

- **The Verbosity** \((n)\) sets a trace level of intermediate output. The default is no intermediate output. If \(n > 0\), then an intermediate statement is printed after computation of every \(n\) columns of the power transition matrix.

- **The Fraction of (Maximum) AU Rate** is a floating point value between \((0, 1.0)\) for determining the range of rates to use in the adaptive uniformization algorithm. The default value is \(0.9\).

Figure 13.8: **Adaptive Transient Solver** ats and available options and parameters.
The output file contains the means and variances of the performability variables. It also contains the following information:

- The rate of the Poisson process used to do the uniformization.
- The number of time points.
- For each time point, the error due to adaptive uniformization, the number of iterations, and the total error.

**Pitfalls and Hints**

- Because this solver is somewhat similar to trs, the pitfalls and hints listed for that solver apply to this solver also.

- Additionally, the transient time points of interest should be short relative to the steady-state time; otherwise, this solver will be inefficient relative to the steady-state solvers or the transient solver trs due to overheads in the adaptive uniformization computation. For example, in the failure-repair dependability model mentioned earlier (see Subsection 13.3.7), the time points of interest should be on the order of the failure times.

- ATS combines the adaptive uniformization (AU) and standard uniformization (SU) algorithms. The idea is that initially AU can be used to make big jumps through activities having low rates to speed up the computation. Eventually, the AU rates converge to the SU rate. Computing AU rates as they converge toward the SU is relatively more expensive as compared to the benefits derived from using AU. By using the parameter Fraction of (Maximum) AU Rate, you can control the range of AU rates that AU will consider. That is, if ATS takes longer to solve a model than TRS, you may want to decrease this parameter in order to lower the overhead of computing the AU rates.
Chapter 14

Connected Model Solver

The connected model editor makes it possible to define a multi-step solution process that generates results by solving a series of separate models and passing results between them. Solutions can be generated from either numerical or simulation-based solvers, or even results from other connected model solvers. Solutions can be computed during the run of the connected model solver, or retrieved from pre-computed results stored in the database. Results can be combined mathematically by equations and algorithms written in connection functions. An example connected model is shown within the connected model editor in Figure [14.1].

14.1 Connected Model Primitives

Connected models are created by graphically representing the solution process as a connected graph of three different types of modeling primitives: solver nodes, database nodes, and connection function nodes. The nodes are connected by directed arcs. When the connected model is solved, each node in the connected graph is processed in turn, receiving results from incoming connections, and passing results to outgoing connections. The output from the final solution step becomes the output of the connected model solver.

The following sections contain descriptions of each of those nodes.

Solver Node  The solver node represents a solution step in the connected model. A solver node can either represent a simulation or a numerical solver. When the connected model is solved, the solver nodes will launch the appropriate solver process. Results that are passed into the solver node override global variable values in the solver node’s child model.
Figure 14.1: An example connected model within the connected model editor.

Figure 14.2: Solver node specification dialog.
14.1. CONNECTED MODEL PRIMITIVES

The solver node is defined using a dialog similar to the one shown in Figure 14.2. The Simulation Model is selected from the list of available solvers using the ... browse button. The Node name defaults to a value based on the solver, but can be changed to a more descriptive name.

The Add Results button is used to specify which results should be exported from this solver node. Results that are exported are displayed in the list at the bottom of the dialog. Users can delete results from the list by selecting them and right-clicking to display a popup menu.

Database Node  The database node specifies a query that extracts previously computed results from the results database. The query is specified using a graphical interface, shown in Figure 14.3. The dialog is used to graphically define the result query by ‘drilling down’ to the appropriate result. The query is defined top down, by first specifying the name of the project and selecting a particular analysis run. As these fields are identified lower fields in the dialog are populated with available options. The result that will be returned is specifying the desired reward name, reward type (mean, variance, distribution), experiment, and time point.

Connection Node  The connection node takes one or more inputs and transforms them to a single output using a custom specified function. The connection function specification dialog is shown in Figure 14.4. The Node Name of this dialog specifies the name of this node. The Set Inputs button is used to activate input variables from the list of available input results. Once activated, the variables appear in the Available Results list. Users can remove items from the list by right-clicking on...
CHAPTER 14. CONNECTED MODEL SOLVER

Figure 14.4: Connection function specification dialog.

the selected item and choosing DELETE from the pop-up menu. The Set Outputs button identifies result names to export from the connection function.

The main feature of the connection node dialog is the Connection Function text area. In this area, Java expressions can be written that map input results to output results. There should be one expression written for each output defined for the connection function node.

The Solve button will execute the function for this connection function node. The Compile button will compile the code written for this connection function, for example, to test the function for syntax correctness.

14.2 File Menu

There are two menu items in the File menu which are specific to the connection editor:

- The Set Iteration Order menu displays a dialog that is used to fine-tune the order that nodes of the connected model graph are traversed. An example of the dialog is shown in figure 14.5. In many cases it is not necessary to make any adjustments with this dialog. Cases that can require user intervention using this dialog include circular graphs and graphs where there might be an implicit dependency between nodes that is not represented by the connection lines.
The First Repeating Node option menu defines the starting node of the solution algorithm. The Nodes list specifies the specific order that the solution nodes will be solved. Use the Move Up and Move Down buttons to change the order of selected entries in the Nodes list.

The final three fields are used to constrain the solution process for fixed-point iterations on cyclic graphs. The first two fields specify the minimum and maximum number of iterations. The final field is used to a stopping criterion, in the form of the relative error between the result from the current and previous iterations.

- The Solve Connection Model menu is used to launch the connected model solution. Selecting this menu option will immediately execute the first node in the connected model graph and begin the process of solving the connected model. The final results are found in the output files of the last solution step executed by the connected model graph.
Part IV

Utilities
Chapter 15

Möbius Data Manager

15.1 Introduction

Möbius provides the ability to store solver results in a PostgreSQL database. Instructions for setting up a database to work with Möbius are detailed in Section C.2. Once the results database has been successfully set up, the user can store any results from simulation and analytical solver executions in it.

Several third-party utilities exist for accessing and managing general data in PostgreSQL databases. PGAdminIII (http://www.pgadmin.org) is a very popular graphical administration tool, and PostgreSQL also distributes a simple text-based command-line tool. However, these tools work on databases without any notion of how or why the data is organized the way it is.

The Möbius Data Manager tool seeks to fill the role of a graphical interface on Möbius results databases in a way that is intuitive and familiar for Möbius users. The Möbius Data Manager allows users to connect to Möbius results databases and manage, organize, and export the data they contain. Möbius users can launch the Möbius Data Manager by clicking on the Utilities→Möbius Data Manager menu item in the Project Manager window.

15.2 Connecting to the Database

When you first launch the Möbius Data Manager, you will immediately be prompted to enter the connection details for the database you’d like to work with. These are the same details you entered in the Database Settings dialog window (section C.2.3).

The Host field should have either the fully qualified domain name or the IP address of the computer running the PostgreSQL database. The Username and
Password fields should contain the PostgreSQL user credentials. The Database field should contain the name of the Möbius results database on the PostgreSQL server.

Once you have entered valid details, click Login to connect to the server and begin the Data Manager session.

To log in to a new connection, you may start the new connection by clicking on the File → New Connection menu item in the Möbius Data Manager.

### 15.2.1 Saved Connection Details

Some connection information can be saved between sessions so that the connection details don’t need to be typed in each time. By default, the Data Manager always stores the details from the most recent connection. The next time the Data Manager is launched, the details from the last session will be automatically filled into the login dialog.

For the sake of security, the Möbius Data Manager never stores passwords. You must always enter your password when using stored connections.

Connection details can also be manually saved with a user-specified connection name. In the Login window, highlight the text in the Connections field and replace it with a name, for example, “Work DB Server.” Fill in the remaining details and then click the Save Connection button. The next time you are prompted to login, the new saved connection details will be in the drop-down list. Selecting it in the list should populate the fields with the saved information.

Should a set of saved connection details no longer be needed, you can easily delete it. Begin by selecting the connection from the drop-down list and click the Delete Connection button to permanently remove it from the list. This does not affect the database server or any data on it.

### 15.3 Navigating the Tree

Once connected to a results database, the Data Manager presents the data as a tree in the upper-left panel of the window. The topmost node represents the database server. The next level represents the projects that have data stored in this database. This level branches based on unique Möbius project names, so two separate projects that are named identically will show up as just one node in the tree. In a situation like that, the results will likely branch apart at lower levels in the tree, but it is typically a good idea to give your Möbius projects unique names.

The third level in the tree represents the solvers in the project. Each node is a conjunction of the solver name and the run name (for analytical solvers, this is
15.3. NAVIGATING THE TREE

the output file name), which are both defined in their respective solver windows in Möbius. The next level represents individual solver executions, which are labeled with the date and time when the run completed.

The fifth level always consists of two nodes: reward variables and experiments. They are simply two ways of grouping the same data. They allow the user to view only data for a specific reward variable or data for a specific experiment. Expanding the reward variables node provides a list of reward variables defined in that solver execution. Expanding the experiments node provides a list of experiments defined in that solver execution.

At the individual solver execution time stamps (fourth level) and all nodes below them, you can view the data associated with the nodes by double left-clicking on them or right-clicking and selecting **Open Table** from the context menu. Depending on the node, certain subsets of the data for the solver execution are presented. For example, if you open the **Experiment 1** node, all the data from Experiment 1, across all reward variables, is presented. If you open the **unreliability** reward variable, all data for that reward variable, across all experiments, is presented. The data is displayed in a table in the right pane.

The columns in the data tables show all the stored attributes for the associated results. Some data cannot be easily displayed with a single value, such as Mean. If you double-click on the value, a dialog appears that states the value, the lower bound, the upper bound, and the bound type for that value. This data can also be exported as described in section 15.5. Global variable parameters, which are defined in Möbius studies, are also displayed as columns on the far right end of the table.

15.3.1 Deleting Data

Every time any solver is run in Möbius while the database feature is enabled, results are added to the database. When refining models and experimenting with different parameters, this can lead to collecting a multitude of unnecessary results in the database. The Data Manager allows users to easily delete as much data as necessary. This **does** delete data from the database itself, so care should be used when deciding which data can be removed.

To delete data, right-click on any node in the tree (aside from the root node) and select **Delete**. This will permanently remove from the database the data associated with the node and all of its children.
15.3.2 Making the Most of the Data Manager Interface

The Möbius Data Manager graphical user interface has been designed from the beginning to be extremely flexible. Any of the panels that have tabs in the upper left corner can be moved, resized, maximized, minimized, stacked, or popped out. This is particularly useful during comparison of results from several runs.

By left-clicking and dragging the tab from one of the table panels, you can move the panel. If you hover over the right side of another panel or the entire window while dragging the tab, a docking arrow and the silhouette of the panel in its prospective position will appear. Table panels can be docked along any edge of any other panel or the main window. There is no limit to the various configurations a user can build by dragging and docking the panels. It is also possible to stack panels by dragging the tabs next to other tabs in a different section. When you drag a tab, a silhouette and a different cursor indicates the prospective position of the tab until you release the mouse button.

You can also pop the data panel out into a new window by either dragging the tab outside of the Data Manager window or right-clicking on the tab and selecting Detached from the context menu. Right-clicking on the tab again and unchecking Detached or dragging the tab back into the Data Manager window will pop it back in.

If a particular data panel requires more space, double-clicking on its tab will maximize it in the Data Manager window and minimize all other panels. Double-clicking the tab again will restore all the panels to their previous state. Maximizing, minimizing, and restoring a panel can also be done by right-clicking on the tab.

Finally, you can close data panels by clicking on the X on the right side of the tab or right-clicking on the tab and selecting Close.

15.4 Data Sets

The Möbius Data Manager allows access to the Möbius results database using an intuitive, tree-based structure. However, viewing data like this requires users to live with the structure defined by the database. To overcome that, the Möbius Data Manager provides a notion of data sets. In the lower left pane of the Data Manager application is the data sets panel. Data sets provide a flexible place to locally gather data from one or more results databases and organize and export the data.

15.4.1 Adding and Removing Data

Begin by clicking on the New button of the Data Sets panel. A new, empty data set is now listed. Next, drag and drop nodes from the tree above onto the data set
15.5. EXPORTING DATA

below. This copies the data from the database and stores it locally in the data set. To view the contents of the data set, double-click on it, and a new data panel will appear.

In the data panel, all the data of the data set is displayed. Heterogenous data is merged as effectively as possible. You can delete rows by highlighting them, right-clicking, and selecting Delete Row from the context menu.

All the currently open data sets are listed in the Data Sets panel. Data sets are saved locally as .mds files. To save a data set, select it from the list and click the Save button. This also allows you to rename the data set, because the name that appears in the list is the same as the filename without the .mds extension.

When you no longer want to work with that data set, you close it by selecting it from the list and clicking the Close button. Should you want to work with it again, click the Open button and locate the file using the open dialog window. All data in the data set is stored locally in the file and can be transferred between machines and users through email or some other method. No database connection is necessary in order to work with the data in a data set, which means that data sets provide an excellent way to work on data in an offline setting.

15.5 Exporting Data

At the individual execution nodes and lower levels, you can export data associated with the nodes. Right-clicking on one of these nodes and selecting Export begins the process. The first window that appears provides a list of available columns for the data. You should check any columns you would like to include in your export and click the Export button.

A file-chooser dialog then appears, and you must provide a file name and path for the export file. The export file is formatted as a comma-separated values file, so it is appropriate to end the filename with the .csv extension, but it is not required. Once exported, the file can be easily used by many applications, such as Microsoft’s Excel and OpenOffice.org’s Calc.

You can also export data from a data set. To begin the process, select the open data set that you want to export and click on the Export button.
Part V

Appendices
Appendix A

Modeling Concepts

Möbius is a software tool to solve for measures of interest of stochastic discrete-event models of real systems. In order to harness the full strength of the tool, one must first understand what the modeling process is; that is, one must know how to specify the model in a particular formalism once the specification of the real system is known. That requires a knowledge of both the system to be modeled and also the formalism in which the system is to be specified. Petri nets, SPNs (stochastic Petri nets), various process-algebra-based formalisms, and GSPNs (generalized SPNs) are examples of modeling formalisms. Möbius specifically supports SANs (stochastic activity networks) which are a generalization of SPN, and have some similarities to the GSPN formalism.

Understanding the dynamics of a stochastic formalism, and hence Möbius, requires a good knowledge of

- Probability theory, including concepts like probability measure, conditional probability, continuous random variables, discrete random variables, PDF (probability density function), CDF (cumulative density function), and commonly-used distributions such as exponential, deterministic, Erlang, and Gaussian, to name a few.

- Stochastic processes, including Markov processes, continuous time Markov chains (CTMC), discrete time Markov chains (DTMC), state transition rate matrices, generator matrices, the Chapman-Kolomogrov equation, steady-state and transient solution, and reward specification.

In the following bibliography, some books are recommended to help you better understand of the concepts mentioned above. Most of the items deal almost entirely with topics related to system modeling and model solution. You therefore
may wish to read some of them in their entirety, to help you grasp a deeper understanding of the inner workings of the tool. However, if you wish to read more selectively, chapters that are particularly relevant to the fundamentals of Möbius are emphasized below.

An Annotated Bibliography


Sections 2.0 to 2.6 contain a nice introduction to probability theory. Sections 3.1 and 3.2 cover commonly-used distributions. Sections 4.0 to 4.4 introduce Markov processes and their properties.


Chapters 1 and 2 provide a somewhat theoretical introduction to Markov processes. Sections 5.1 and 5.2 introduce Petri nets, and Chapter 8 introduces timed Petri nets. Section 10.1 covers quantitative analysis of GSPNs.


Sections 6.1, 6.2, 6.3, 6.6, and 6.8 contain a mathematically rigorous introduction to CTMCs and their properties.


This is a classic book on simulation and analysis of simulation results. In particular, Chapters 1 and 2 contain a nice introduction to probability theory.

Chapter 1 introduces modeling processes and Sections 2.1 and 2.3 introduce GSPNs.


Chapter 1 and Sections 2.1 to 2.4 introduce Petri nets. Sections 3.1 to 3.3 introduce timed Petri nets. Chapters 5 and 6 introduce GSPNs and their properties and analysis.


Chapters 1 and 2 introduce probability theory, and Chapter 5 introduce Markov chains.


This is a well-written book on probability and Markov chains containing a lot of examples. Chapters 1, 2, and 4 and Section 5.1 introduce probability theory, Sections 6.1 and 6.2 cover Poisson processes, and Sections 8.1 to 8.4 discuss Markov chains and their properties.


This is a classic, widely cited book on the subject. It gives in-depth understanding of how some of the solvers in Möbius work.

This is a strongly recommended, comprehensive, and well-written book that covers a wide variety of material required for sound understanding of the mathematical concepts used in modeling stochastic systems. The following chapters are particularly useful: Chapter 1, Sections 2.1 to 2.4, 2.9, 3.1 to 3.5, 3.8, 3.9, 4.1, 4.2, 4.6, 6.1 to 6.4, and Chapter 8
Appendix B

Example: Fault-Tolerant Multiprocessor System

This appendix presents an example of a system that can be modeled using Möbius. It starts with a description of the system, and then guides you through one way to build a model of the system and solve it using both simulation and numerical solution. The example is intended to take you step-by-step through the process of creating and solving a model in Möbius, and to exhibit many of the capabilities and features of the tool.

B.1 System Description

The system under consideration is a highly redundant fault-tolerant multiprocessor system adapted from [20] and shown in Figure B.1. At the highest level, the system consists of multiple computers. Each computer is composed of 3 memory modules, of which 1 is a spare module; 3 CPU units, of which 1 is a spare unit; 2 I/O ports, of which 1 is a spare port; and 2 non-redundant error-handling chips.

Internally, each memory module consists of 41 RAM chips (2 of which are spare chips) and 2 interface chips. Each CPU unit and each I/O port consists of 6 non-redundant chips. The system is considered operational if at least 1 computer is operational. A computer is classified as operational if, of its components, at least 2 memory modules, at least 2 CPU units, at least 1 I/O port, and the 2 error-handling chips are functioning. A memory module is operational if at least 39 of its 41 RAM chips, and its 2 interface chips, are working.

Where there is redundancy (available spares) at any level of system hierarchy,
there is a coverage factor associated with the component failure at that level. For example, following the parameter values used by Lee et al. [20], if one CPU unit fails, with probability 0.995 the failed unit will be replaced by the spare unit, if available, and the corresponding computer will continue to operate. On the other hand, there is also a 0.005 probability that the fault recovery mechanism will fail and the corresponding computer will cease to operate. Table B.1 shows the redundant components and their associated fault coverage probability. Finally, the failure rate of every chip in the system, as in [20], is assumed to be 100 failures per billion hours.\footnote{0.0008766 failures per year.}

Table B.1: Coverage probabilities.

<table>
<thead>
<tr>
<th>Redundant Component</th>
<th>Fault Coverage Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM Chip</td>
<td>0.998</td>
</tr>
<tr>
<td>Memory Module</td>
<td>0.95</td>
</tr>
<tr>
<td>CPU Unit</td>
<td>0.995</td>
</tr>
<tr>
<td>I/O Port</td>
<td>0.99</td>
</tr>
<tr>
<td>Computer</td>
<td>0.95</td>
</tr>
</tbody>
</table>
B.2 Getting Started

A model of the system in this example is included with the Möbius distribution. Refer to Section [C.1](#) for instructions on installing the example models. You are encouraged to open the model and follow the detailed discussions of its various components in the sections below.

From the Möbius Project Manager window, click **PROJECT→UNARCHIVE**. A dialog will present a list of archived projects in the project directory. Choose **Multiproc-Paper** and hit **Unarchive**. After the project has been successfully unarchived, you will be prompted to resave the project using **PROJECT→RESAVE**. At the dialog, choose **Multiproc-Paper** again, hit **Resave**, and wait until all components have been built. The **Multiproc-Paper** project editor will appear as shown in Figure [3.1](#).

B.3 Atomic Models

To build a model for an entire system, begin by defining SAN submodels to represent the failures of various components in the system.

The SAN submodel of the CPUs is called cpu_module and is shown in Figure [B.2](#). To open this model, click the **Atomic** tab in the project panel, and then double-click on cpu_module or right-click on it and select **OPEN**. The places named cpus and computer_failed represent the current state of the CPUs and the current state of the multiprocessor system, respectively. That is, the number of tokens in cpus represents the number of operational CPUs in a given computer. Likewise, the number of tokens in computer_failed indicates the number of computers that have failed in the system. To open any of these places, right-click on the place and select **EDIT**. This will bring up the **Place Attributes** dialog, in which you can edit the **Name** of the place and the initial marking (number of tokens) of the place. Note that the **Tokens** field can be specified with either a constant or a global variable name. For example, the place cpus has been initialized with three tokens, as each computer consists of three CPU units.

To create a new place, either click the blue circle icon in the toolbar or select **ELEMENTS→PLACE** from the menu. Then click where you would like the place to go in the editor. The **Place Attributes** dialog will appear, and you can edit the **Name** of the place as well as the initial marking of the place in the **Tokens** field, as described earlier. To delete a place, right-click on it and select **DELETE**, and hit **OK** to confirm.

The places labeled ioports, errorhandlers, and memory_failed are also included in this model to aid in reducing the size of the state space for the overall system.
model by lumping as many failed states together as possible. Additional state lumping (beyond that provided by the reduced base model construction method) can be achieved because once a computer fails, there is no need to keep track of which component failure caused the computer failure. More specifically, because of the assumption that all internal components of the failed computer have failed, the states that represent a computer failure due to a failure of a CPU unit, a memory module, an I/O port, or an error-handling chip are combined into a single state. The marking of the combined state is reached by setting the number of tokens in each of the places cpus, ioports, and errorhandlers to zero, setting the number of tokens in memory_failed to 2, and incrementing the number of tokens in computer_failed.

The failure of a CPU unit corresponds to the completion of timed activity cpu_failure. To open this activity, right-click on it and select Edit. This will bring up the Timed Activity Attributes dialog. In this dialog, you can edit the name of the activity and the distribution of its firing delay in the Time distribution function field. For this activity, the Exponential distribution should be selected. The activity completion rate is shown in Table B.2. This rate corresponds to $6 \times$ times the failure rate of a chip times the number of operational CPU units in the computer. If a spare CPU unit is available (i.e., cpus->Mark() == 3), three cases are associated with the activity completion, as designated in the Case quantity field. To define the case probabilities, click on the appropriate case number’s tab and type the expression in the box. The expression for the case probability can be a constant, a global variable, or a C++ statement returning a value as in this example. The first case represents a successful coverage of a CPU unit failure. If that case occurs, the failed CPU unit is replaced by the spare unit, and its corresponding computer continues to operate. The second case represents the situation

\[\text{Remember that each CPU unit consists of 6 non-redundant chips.}\]
in which a CPU unit failure occurs that is not covered, but the failure of its corresponding computer is covered. If that happens and a spare computer is available, the failed computer is replaced by the spare computer and the system continues to operate. However, if no spare computer is available, the multiprocessor system fails. The third case represents the situation in which neither the CPU failure nor the corresponding computer failure is covered, resulting in a total system failure.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpu_failure</td>
<td>$\text{expon}(0.0052596 \times \text{cpus-&gt;Mark}())$</td>
</tr>
</tbody>
</table>

On the other hand, if no spare CPU is available (i.e., $\text{cpus->Mark}() == 2$), then a CPU unit failure causes a computer failure. In this marking, two possible outcomes may result from the completion of activity cpu_failure. In the first, a spare computer is available, so that the computer failure can be covered. In the second, no spare computer is available, and system failure results. Table B.3 shows the case numbers and the probabilities associated with each case for the activity cpu_failure. It is clear that the case probabilities are marking-dependent, since the coverage factors depend on the state of the system.

The input gate Input_Gate1 is used to determine whether the timed activity cpu_failure is enabled in the current marking, and hence can complete. The cpu_failure activity is enabled only if at least 2 working CPU units are available and their corresponding computer and the system have not failed. Table B.4 shows the enabling predicate and function associated with this gate.

The output gates OG1, OG2, and OG3 are used to determine the next marking based on the current marking and the case chosen when cpu_failure completes. They correspond to the different situations that arise because of the coverage or non-coverage of system components. Table B.5 lists the output gates and the function of each gate.

In a SAN model, relationships between elements are designated by connecting lines or arcs. For example, places and input gates may be connected to an activity to indicate they are enabling conditions for the activity. An activity (or one of its cases) may be connected to a place or an output gate to indicate that upon completion of the activity, the marking of the place is affected or the output gate function is executed. It is not necessary to connect an output gate to a place whose marking the output gate function changes. Such a connection exists only to ease understanding of the model. To draw a connecting line or arc, choose ei-
Table B.3: cpu_module case probabilities for activities.

<table>
<thead>
<tr>
<th>Case</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cpu_failure</td>
</tr>
</tbody>
</table>
| 1    | if (cpus->Mark() == 3)  
|      |     return(0.995);  
|      |     else  
|      |     return(0.0);  |
| 2    | if (cpus->Mark() == 3)  
|      |     return(0.00475);  
|      |     else  
|      |     return(0.95);  |
| 3    | if (cpus->Mark() == 3)  
|      |     return(0.00025);  
|      |     else  
|      |     return(0.05);  |

Table B.4: cpu_module input gate predicates and functions.

<table>
<thead>
<tr>
<th>Gate</th>
<th>Enabling Predicate</th>
<th>Function</th>
</tr>
</thead>
</table>
| Input_Gate1  | (cpus->Mark()>1) & &  
|              | (memory_failed->Mark()<2) & &  
|              | (computer_failed->Mark()<num_comp)                      | identity |
Table B.5: cpu_module output gate functions.

<table>
<thead>
<tr>
<th>Gate</th>
<th>Function</th>
</tr>
</thead>
</table>
| OG1  | if (cpus->Mark() == 3)  
|      | cpus->Mark()--; |
| OG2  | cpus->Mark() = 0;  
|      | ioports->Mark() = 0;  
|      | errorhandlers->Mark() = 0;  
|      | memory_failed->Mark() = 2;  
|      | computer_failed->Mark()++; |
| OG3  | cpus->Mark() = 0;  
|      | ioports->Mark() = 0;  
|      | errorhandlers->Mark() = 0;  
|      | memory_failed->Mark() = 2;  
|      | computer_failed->Mark() = num_comp; |

Another way to model the failure of CPU modules would be to model the failure of a single CPU module as a SAN and replicate this model three times. However, since the failure of any chip inside the CPU module causes the CPU to fail, and each chip is assumed to have an exponentially distributed failure rate, the failure rate of one CPU module is just the sum of the failure rates of the 6 CPU chips. Therefore, modeling the failure of one CPU module, and then replicating this model three times, results in a model that is equivalent to the cpu_module...
The SAN submodels of the I/O ports, the memory module, and the two error-handling chips are shown in Figures B.3, B.4, and B.5, respectively. The line of reasoning followed in modeling each of these components is similar to that followed in modeling the CPU modules. Note the similarity between the io_port_module and cpu_module SANs. A more detailed discussion of creating SAN models can be found in Section 5.1.
B.4 Composed Model

Now the replicate and join operations previously defined (see Section 6.1) are used to construct a complete composed model from the atomic models. Figure B.6 shows the multi_proc composed model for the multiprocessor system. To open this model click the Composed tab in the project panel, and double-click on multi_proc or right-click on it and select OPEN.

The leaf nodes represent the individual submodels, or atomic models, that were defined in the previous section. The memory_module is replicated 3 times, corresponding to the number of memory modules in each computer, with the places computer_failed and memory_failed (see Figure B.4) held in common among all the replicas. You can see where that is set by right-clicking on the Rep node whose child is the memory_module submodel, and choosing EDIT. The Define Rep Node: REP1 window will appear. Here the name of the Rep node is specified as
Repl, and the Number of Reps is specified as the global variable \texttt{num.mem.mod}, which is later defined to be 3 in Section B.6. The two lists Unshared State Variables and Shared State Variables define which state variables are shared, or held in common, among all replicas. To move a state variable from one list to the other use either the Share $>$ or $<$ Unshare button. To move all state variables use the Share All $>>$ or $<<$ Unshare All button. You can create a new Rep node by selecting the red \texttt{R} icon from the toolbar or choosing ELEMENTS $\rightarrow$ Rep from the menu. Then click inside the editor where the Rep node is to be placed and specify the name of the node and the number of Reps in the Define Rep Node dialog. A Rep node must have as its child either an atomic model or another composed model. Click on the black \texttt{S} icon in the toolbar or select ELEMENTS $\rightarrow$ SUBMODEL to add a submodel. Then you can draw a connecting line from the Rep node to the child submodel in the same way that you would draw connecting lines in the atomic model editor (see Section B.3). Once a Rep node is given a child, the shared state variables can be defined by editing the Rep node again.

The three memory modules are then joined to the I/O ports model (Figure B.3), CPUs failure model (Figure B.2), and error-handler model (Figure B.5) to form a model of a computer. In the Join node, places with a common name are shared, and thus treated as single places among all system submodels. To open this node, right-click on the blue Join node and select EDIT. This will bring up the Define Join Node Dialog. Here, the Join node name is specified as Join1 and shared state variables can be created. The Join State Variables list shows all state variables that are shared across multiple submodels in the Join. Clicking on a shared variable in this list will display the corresponding name of the shared variable in each of the submodels among which it is shared under the Submodel Variables list. The \# Shared column indicates how many submodels share each Join state variable. To share a state variable among submodels in a Join, click the Create New Shared Variable button, give a name for the new variable, and select the submodel state variables that are to be shared. In this example, places with a common name across different submodels are shared; this is achieved with the Share All Similar Variables button. A new Join node can be created by clicking on the blue \texttt{J} icon in the toolbar or selecting JOIN from the ELEMENTS menu. Then the Join node must be connected to its children nodes with arcs as discussed previously. A Join node can have as its children submodels, Rep nodes, or other Join nodes.

Finally, the joined SAN model of one computer is replicated \texttt{num.comp} times by the ‘Rep2’ node to generate the complete model of the multiprocessor system. More information about creating composed models and the composed model editor can be found in Chapter 6.
B.5 Reward Variables

After the composed model of the multiprocessor system has been built, the next step in the model construction process is to define reward variables. Reward variables permit us to compute interesting measures from the model. This example, for instance, focuses on measuring the reliability of the multiprocessor system over a 20-year mission time. The system is considered unreliable by time $t$ if all of the $num\_comp$ computers in the system have failed. In terms of this model, the system is unreliable when there are $num\_comp$ tokens in place computer_failed.

To define the reliability variable, click on Reward in the project panel, then click NEW (either in the toolbar or by right-clicking on Reward and selecting NEW) and specify the new performance variable model name. Or, to view the existing performance variable model for this example, click the Reward tab in the Project panel. All previously defined variables are listed under this tab. The reliability variable should already have been defined, and you can open it for revision either by double-clicking on the variable MultiProc_PV or by choosing it and then clicking on the Open button on the panel. That will open up the Reward Editor for the variable.

On the left-hand side of the Reward Editor window, there is a Variable List sub-window containing all defined reward variables for this model. In the example, unreliability is the only variable. Choose it for revision by clicking on it once. Then click on the Submodels tab to choose the submodels on which the reward is to be computed. Because unreliability is defined on the place computer_failed in the submodel cpu_module, choose this submodel by clicking on it once (see Figure B.7).

Next, to define the rate reward for unreliability, click on the tab Rate Rewards. This will bring up two sub-windows. The top sub-window lists all available state variables in the model on which rate rewards can be defined. The bottom sub-window, Reward Function, is a text area for entering C++ code for computing reward for the currently selected reward variable (see Figure B.8). In this example, a reward of $(1/num\_comp)$ should be returned when all of the computers have failed, because the reward is evaluated over all submodels in the composed model. That is, a reward of $(1/num\_comp)$ is accumulated once for each computer, or a total of $num\_comp$ times, for a total reward of 1. Thus, the reward for a state in which all computers have failed is 1, and the mean unreliability of the system (for example) can be found by calculating the mean of this reward variable. The C++ code that should be entered in this sub-window is

if (cpu_module->computer_failed->Mark() == num_comp)
{

return 1.0/num_comp;
}

Now click the Simulation tab to view the parameters for simulation. Since the goal is to measure the unreliability of the system at a particular time (20 years), the Type has been set to Instant of Time and the Start time to 20.0 as in Figure B.9. You can ignore the Estimation and Confidence tabs for now.

At this point, if you wanted to define impulse rewards on your model, you could do it here. While rate rewards are evaluated when the model is in a particular state, impulse rewards are evaluated upon action firings. Click on the Impulse Rewards tab to view a list of available actions in the model. The three cases of activity cpu_failure in cpu_module will be listed, accompanied by a column indicating whether impulse rewards have been defined on any of them. Since there are no impulse rewards in this example, this column should read No for each activity. To define an impulse reward on an activity, click on the appropriate activity in the Action Name column and type the reward function in the Impulse Reward Function text box. For example, to count the number of times the cpu_failure activity fires during the first 20 years, the impulse function should return 1 for each of the three cases. Under the Simulation tab, you would set the Type to Interval of Time with a Start of 0.0 and Stop of 20.0. For the purposes of this example, however, do not define impulse rewards. More information about the Reward Editor can be found in Chapter 7. Save the reward variable definition...
B.6 Study and Experiments

Once all of the variables of interest have been defined as described in the previous section, you can create a study, or set of experiments, to evaluate the model of the multiprocessor system. To do so, simply define values for the global parameters of the model. To begin, click on the Study tab in the Project panel. There is a pre-defined study, vary_num_comp, for this example. Open vary_num_comp for revision either by double-clicking on it or by choosing it and clicking on the Open button on the Project panel. This will bring up the Study Editor, in which the global parameters for the model are defined, as shown in Figure B.10.

To modify the parameters that have numerical values in the column Variable Value in Figure B.10 click directly on the values. For the other parameters that have labels, such as Incremental Range, Functional Range, Manual Range, or Random Range, you can modify the values by selecting the variable and then clicking on the corresponding button at the bottom of the editor window. For instance, the variable num_comp has an incremental range from 1 to 3 with increments of 1. You can change its values by selecting it and then clicking on the Incremental Range button. This brings up the editor window as shown in Figure B.11. As shown in the figure, the variable ranges from 1 to 3 with additive
increments of 1. You can reveal all the values in the range by pressing the button View Values.

From the parameter values, three experiments have been created. The experiments differ in the values for the parameter num_comp. You can individually activate or deactivate the experiments by pressing the button Experiment Activator. Pressing this button brings up a window similar to the one shown in Figure B.12. Note the checkboxes in the row Active. Active experiments are indicated by the check marks. Deactivate experiments by clicking on these checkboxes to remove the check marks. For the example, leave these experiments activated and accept the existing values by pressing the button OK. This brings you back to the Study Editor. Save the values with FILE→SAVE.
B.7 Solving the Model

After the studies have been created, the next step in model construction is to solve the model for the measures of interest. Möbius provides two methods for solving models: numerical analysis and simulation. For this example, either method can be used to compute the transient solution of the model. The next subsection describes numerical analysis using a transient solver. The following subsection will describe the procedure for conducting simulation.

B.7.1 Numerical solvers

Section B.5 of this chapter described how reward variables are defined to allow computation of interesting measures in the model. The reward variable defined there is the unreliability of the multiprocessor system for a 20-year mission time. You can now compute the transient solution of the unreliability of the system.

Before the unreliability measure can be computed, the state spaces of the model must be generated. The state space is a description of all the states that the model may visit and reside in. To generate the state spaces, first open the state-space editor by clicking on the Solver tab in the Project panel. Then double-click on the tab ssg to open the State Space Editor for a previously created state space. That should bring up a window similar to the one shown in Figure B.13(a).

There are a few things to note on this editor. The study vary_num_comp is the study defined in the previous section, and the state spaces that will be generated are based on the global parameters defined in vary_num_comp. There are three experiments for this study, as shown in the Experiment List, and three corresponding
state spaces will be generated for them. You can prevent the state spaces for these experiments from being generated by deactivating the corresponding experiments. Do so by pressing the button Experiment Activator. This will bring up a window similar to the one in Figure B.12. For this example, disable Experiment 3, because the computation time for it may be lengthy on machines without enough resources. Finally, press the button Start State Space Generation to generate the state spaces for the experiments.

As the state spaces are being generated, you can check their progress by pressing the tab SSG Output. This should show a window similar to the one in Figure B.13(b). Note that in the figure, Experiment 2’s state space was generated with 10,114 states. At any time, you can stop the state-space generator by pressing the Stop button. When the two state spaces have been generated, save them and close the editor with FILE→SAVE and FILE→CLOSE.

After the state spaces have been generated, the next step is to use a transient solver to compute the unreliability measure based on these state spaces. Möbius provides several transient solvers, but this example will demonstrate the transient solver based on standard uniformization. To create a transient solver, select the Numerical tab and click the New button. This brings up a menu of solvers from which you can choose. Select Transient Solver from this menu and type the name trs in the text box Numerical Name. Next, press OK to select the state spaces on which the transient solver will be used. Select ssg from the menu presented
B.7. SOLVING THE MODEL

(a) The parameters for the state-space generator.

(b) The output of the state-space generator.

Figure B.13: State-space generator editor.

and press OK to bring up the transient solver editor, which will be similar to the one shown in Figure B.14(a). In the transient solver editor, enter 20.0 in the text box Time 1 to indicate that the time point of interest is the twentieth year. If there is a file name in the text box Output File Name, erase it so that the results are displayed on the screen. Finally, press the button Solve to start the transient solver.

After the transient solver has completed, you can display the results by pressing the Output tab. Figure B.14(b) shows the results from running the transient solver.

Note that the figure shows the output for Experiment 2, which corresponds to the state space for the multiprocessor system that has 2 computers (that is, num_comp = 2). The unreliability by the twentieth year is 0.017465.

B.7.2 Simulation

The numerical results obtained through the transient solver may be verified using discrete-event simulation. First, open the simulation editor by double-clicking on sim under the Solver tab in the project editor. This will bring up the window shown in Figure B.15(a). This window presents the parameters for the simulation. The Current Study text box specifies the child study, which in this case is vary_num_comp. Note that the Simulation Type selected is Terminating Simulation. That means that you will obtain a transient solution, which is appropriate for the length of time you are studying (20 years). Möbius knows which type of simulation you are running because in the reward editor, unreliability is specified
APPENDIX B. EXAMPLE: FAULT-TOLERANT MULTIPROCESSORSYSTEM

(a) Transient solver input parameters. (b) The output from running the transient solver.

Figure B.14: Transient Solver editor.

under the Simulation tab to be an Instant of Time performance variable with a start time of 20.0 (see Figure B.9). In the Maximum Batches text box, type 100000. This number sets the maximum number of batches to run for each experiment. The actual number of batches run depends on how long it takes the results to converge to the specified confidence interval. In this example, the default number of batches is not enough to allow the results to converge. The remaining default options (random number options, build type, run name, and so forth) should suffice for now. For more information on these options, consult Chapter 12.

The Network Setup tab shown in Figure B.15(b) displays the list of machines available for running a distributed simulation. Your local machine should be listed on the right under Selected Systems. Any other machines available will be listed under Available Systems. To add a network machine to the list of available systems, click the Edit Machine/Group Info button to bring up the Network Machine and Group Settings dialog. To add an available machine to a distributed simulation, select it in the left list and click the > button to move it to the Selected Systems list.

Click the Run Simulation tab and then the Start Simulation button. This begins the process of compiling the models and linking with the Möbius simulator libraries. The window (which appears in Figure B.16(a)) displays the status of the running simulation.

The output of the simulation can be found under the Simulation Info tab shown
B.7. SOLVING THE MODEL

(a) Simulation parameters for Multiproc-Paper.

(b) Network setup for simulation.

Figure B.15: Simulation input and network configuration windows.

(a) Running simulation.

(b) Simulation results.

Figure B.16: Simulation status and output windows.
in Figure B.16(b). The table at the top of the dialog shows the status of each active experiment (i.e., whether it is finished, running, or waiting to run), as well as the number of CPUs dedicated to the simulation and the number of batches completed. Click on any experiment to view its status in more detail below. A progress bar, indicating how near the experiment is to completion, is displayed, along with the elapsed running time. The mean value for the performance variable unreliability is shown, along with its confidence interval, as the simulation progresses. A number in red indicates that the value has not yet converged, while a blue number indicates that the value has converged to the specified confidence interval. Finally, click the Show Results button to bring up a window containing more detailed simulation results. The mean unreliability over the 20-year mission time is 0.016895 for Experiment 2, which corresponds to the multiprocessor system consisting of two computers. Observe that the unreliability obtained through numerical solution falls within the confidence interval for the mean obtained through simulation, and thus the results from the two solution techniques match.
Appendix C

Software Installation

C.1 Example Models

Some example models are included in the Möbius installation. They are located in <path>/Mobius-2.4/examples. Each time a new Möbius project directory is created, Möbius will ask if the example models should be installed. Answering “yes” will copy them into the new project directory.

You may also copy them manually into existing project directories. Simply copy the desired example model tar file into your Möbius project directory (or make a symbolic link on Unix). Then unarchive it using the UNARCHIVE option from the Möbius project manager file menu. After it is unarchived, you must resave it using the RESAVE menu in the Project Manager. The resaving step generates the appropriate C++ files and model libraries for the project.

C.2 Database Installation

In this section, we present two methods to install a Postgresql database for use with Möbius. This allows the user to use advanced Möbius features like plotting, design of experiments, and connection.

C.2.1 Windows instructions

Under Windows, we describe a method to install a Postgres database using the Windows native installer package.

2. Edit the file `pg_hba.conf` from the Start Menu link.

   Make sure the three similar lines at the bottom of the file contain:

<table>
<thead>
<tr>
<th>Local</th>
<th>All</th>
<th>All</th>
<th>127.0.0.1/32</th>
<th>md5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>All</td>
<td>All</td>
<td>::1/128</td>
<td>md5</td>
</tr>
</tbody>
</table>

   This allows all local connections as well as IPv4 and IPv6 connections and authenticates using md5. This may not be secure enough for your facilities depending on your network configuration. Please consult your System Administrator and refer to http://www.postgresql.org/docs/8.1/interactive/client-authentication.html for more advanced configurations.

3. Start the database service by clicking the “Start Service” link in the Start Menu.

4. Test the connection to the database. From a cmd shell, type:

   `psql template1 -U postgres`

   It should prompt your for the password for the user postgres. The default password is `postgres`. A list of menu options should appear if the connection has been established.

5. Create a user.

   From the SQL shell established in the previous step, create a new user with the following command:

   `CREATE USER 'user' WITH CREATEDB PASSWORD 'password';`

   where “user” is the username and “password” is the password. Exit the SQL shell by typing “\q”.

6. Test the user connection.

   From a cmd shell, type “`psql template1 -U 'user'`”, where “user” is the previously created user. Type in the password when prompted. If successful, exit with “\q” and proceed to set up Möbius.
C.2. DATABASE INSTALLATION

C.2.2 Ubuntu Instructions

In this section, we describe a method to install a PostgreSQL database in Ubuntu to work with Möbius. The installation instructions are specific to the Ubuntu distribution, but are general enough that they can be used for most distributions as well. For more information, refer to the Ubuntu PostgreSQL installation documents at https://help.ubuntu.com/community/PostgreSQL.

1. Install PostgreSQL server, client, and admin tools

   $ sudo apt-get install postgresql postgresql-client pgadmin3

2. Set the postgres database password

   $ sudo -u postgres psql postgres
   \password postgres

C.2.3 Möbius Installation

1. From the Project Manager, select PROJECT→PREFERENCES.

2. Select the Database preference page in the left column.

3. Check the box next to Database Enabled.

4. Type the user name and password created in the previous sections.

5. Type a name for database to store the information. Different databases can be used for different models or projects to separate data.

6. If the database is installed locally, leave the database server name and port fields alone. If it is installed remotely, enter the hostname and port of the installation host.
Bibliography


Index

absorbing state, 162, 165, 198, 200
   in dss, 198
   in iss, 200
action, 3, 5, 7, 9
   deterministic, 160, 192, 204, 207
   209
distribution, 9, 101, 157, 160, 192
   marking-dependent, 206
in composition, 21
in numerical solution, 192–194, 204
in rewards, 22, 99–101
in SANs, 49, 50
in SSG, 159–161, 164
instantaneous, 9, 160
last firing time of, 67
timed, 9, 160, 192
action synchronization formalism, 94
activation predicate, 51
active experiment, 169, 172
activity, 19, 50, 181, 182, 236, 237
   arcs, 66
cases, 51, 64, 237
collection, 49–52, 181, 236, 237
delay distribution, 50, 51, 64, 237
   enabling, 49, 51, 52, 181
   in editor, 57
   instantaneous, 50, 51, 57, 63, 64
   reactivation, 51
   specification, 63, 64
timed, 50, 57, 63, 64, 236, 237
type of action, 7

with user-defined functions, 47
add cache size, 175
aggregation matrix, 201
arc
directed, 52
archive, see project, archive
array(s), 47, 165, 182, 188
ASCII, 165, 182, 189
csv file, 182, 188
atomic formalism, 52, 82, 84
atomic model, 3, 5, 7, 8, 20, 22, 37, 38
   45, 49, 57, 84, 86, 91, 101, 105
   165, 172, 235, 241
available systems, 183, 250
batch means, 102
   observations, 23, 102, 103
batches, 180–182, 186, 188, 189
   250, 252
binary, 182, 185, 186, 189
   dat file, 182, 186
bucket, 68, 70
   Buckets, 70
   Buckets and Balls, 68, 70
   Arcs and Lines, 74
   Elements, 70
   transition, 68, 68
      instantaneous, 68
      timed, 68, 70, 71
buckets and balls, 49, 100
build architecture, 181
build type, 161, 169, 170, 181, 250
case probability, 51
marking-dependent, 237
child, 169
child model, 38
client machine, 185
command line
running flat state space generator from, 164
running simulator from, 186
running symbolic state space generator from, 174
composition
architecture, 32
in flat state space generator, 164
in simulator, 181
mode, 30
of model code, 5
optimizations, 32
options, 177
compile, 170
composed model, 84
composed model, 83
editor, 88
composition formalism, 21
compositional lumping, 173
compositional lumping, 167
compositional lumping, 167
confidence interval, 180
confidence interval, 186
connected model solver, 215
constraints
constraints checker, 40
continuous time Markov chain, 166
CTMC, see continuous time Markov chain
experiment, 22
data structure, 166
symbolic, 166
vs. sparse matrix representation, 166
data structures
symbolic, 171
database, 32
debugging, 170
distribution, 9
beta, 13
binomial, 10
Conditional Weibull, 18
deterministic, 11
distribution functions, 67
Erlang, 12
exponential, 11
gamma, 11
geometric, 15
hyperexponential, 14
lognormal, 19
negative binomial, 15
normal, 18
triangular, 17
uniform, 16
Weibull, 17
documentation
of model, 40
of project, 29
PDF, 29
PNG, 29
SVG, 29
drop tolerance
in dss, 197
Enable Compositional Lumping, 171
events, 78
execution policy editor, 65
activator, 112, 160, 161
active, 112, 160, 163, 246, 252
inactive, 112
extended place, 46, 53, 57, 64, 66
Fault Tree, 78, 80
fault tree, 49, 77, 78
fill-in
in diss, 206, 209
in dss, 198
flat state space generator, 159, 159, 161, 164, 165, 168, 169, 204
command line options, 164
limitation of, 166
vs. symbolic state space generator, 167
full archive, see project, archive, full archive
functional range, 106, 107, 107, 108, 245
global variable, 22, 44, 44, 45, 58, 60, 61, 63, 64, 91, 105, 109, 111, 163, 172, 179, 195
reported in solvers, 195
graph composition formalism, 92
grid setting, 56, 57
hash table, 171
help
project manager
help menu, 35
IAD, see Iterative Aggregation/Disaggregation
impulse reward, see reward, impulse reward
in symbolic state space generator, 168
incremental range, 106, 106, 107, 109, 245
initial marking
for transient solution, 211
initial partition, 173
input gate, 49, 51, 51, 52, 57, 60, 61, 66, 237, 238
enabling predicate, 51
function, 52
specification, 60
installation path, 34
instantaneous transition, see Buckets and Balls, transition, instantaneous, 70, 73
interval of time, 101, 102, 192
Iterative Aggregation/Disaggregation, 201
iterative steady state solver, 168
jackknife variance calculation, 182
Jacobi
in iss, 198
Jacobi solver, 168
join node, 84, 86, 88, 88, 92
child, 21, 84, 84, 88, 90
lagged Fibonacci, 180, 189
local exploration cache size, 175
logic gates, 78
LU decomposition
in dss, 196
lumped CTMC, 167
lumped state space, 84, 171
lumping, 167
lumping algorithms, 173
manual range, 106, 109, 109, 245
mapping MDD, 173
marking(s), 49, 52, 58, 60, 63, 64, 192, 206, 209, 211, 236, 237
Markov process, 1159, 211
matrix diagram, 167, 173
MDD, see multi-valued decision diagram
MDD unique table size, 171, 175
mean, 20, 23, 50, 64, 102, 104, 186, 193, 212, 252
minimal archive, see project, archive, minimal archive
model-level lumping, 173
model-level lumping, 167, 168
MTMDD, see multi-terminal multi-valued decision diagram
multi-terminal multi-valued decision diagram, 173
multi-terminal multi-valued decision diagram, 167
multi-valued decision diagram, 167, 172
MxD, see matrix diagram
MxD unique table size, 171, 175
NCD, see Nearly Completely Decomposable
Nearly Completely Decomposable, 203
network
machine, 54
setup, 183
nodes, 78
normal mode, 169
normal mode, 161, 165, 164, 169, 170, 174, 181
compilation in, 161
numerical solver
iterative steady-state solver (iss), 201
numerical solver, 157, 191, 194, 247
accumulated reward solver (ars), 211
pitfalls and hints, 212
adaptive transient solver (ats), 193
pitfalls and hints, 212
advanced deterministic iterative steady-state solver (adiss), 206
for interval-of-time measures, 209
marking-dependent deterministic times, 209
applicability for different measures and models, 193
child of, 191
common options, 194
deterministic iterative steady-state solver (diss), 204
for interval-of-time measures, 206
marking-dependent deterministic times, 206
pitfalls and hints, 206, 209
direct steady-state solver (dss), 196
pitfalls and hints, 198
iterative steady-state solver (iss), 198
pitfalls and hints, 200
Takahashi steady-state solver (tss), 201
pitfalls and hints, 203
transient reward solver (trs), 210, 213, 214, 247, 249
pitfalls and hints, 211
optimizations, 170
optimize mode, 161, 169, 171
compilation in, 161
output gate, 39, 52, 57, 61, 63, 66, 237, 239
function, 52
specification, 62
PEPA, 2, 49, 75
Choice, 76
choice operator, 76
Cooperation, 76
Formal Parameters, 75
Guards, 75
Hide, 76
PEPA, 75
Prefix, 76
INDEX

Process Variable, 75
shared activities, 76
performance variable, 173
performance variable, see reward variable, 172, 173
performance variables, 167
phases of symbolic state space generation, see symbolic state space generator, phases of
place, 5, 7, 21, 46, 49, 52, 57, 58, 60, 61, 63, 64, 66, 67, 78, 83, 100
plot complementary distribution, 195
preferences, 32
probability distribution vector, 168
probability distribution vector, 166
process algebra, 75
project, 28, 35, 37, 37
archive, 29, 235
full archive, 29
minimal archive, 29
clean, 29
closing an open, 40
compiling, 34
copy, 28
copying a model to a different, 39
copying a model within a, 39
creating a new, 28
creating a new model in, 38
delete, 29
deleting a model from, 40
documentation, see documentation, of project
documentation, see documentation, of model
opening an existing, 28, 29
operations on, 38
project directory, 29, 35, 40, 40, 188, 235, 253
UltraSAN root, 39
project editor, 37, 38, 191
project manager, 27, 235, 253
project menu, 28
project tree, 38, 40
children in, 38, 85, 93
projects directory, 35
renaming an existing, 29
tools, 27
unarchive, 29, 34, 235, 253
UltraSAN project, 39
random number
generator, 179, 180, 189
seed, 180, 189
random range, 106, 109, 109, 110, 245
range study, 105, 105, 106, 110
rate reward, see reward, rate reward
reachable states, 167
reactivation
function, 51
predicate, 51, 160
replicas, 91, 91, 92, 180, 183, 186, 241
replicate (rep) node, 84, 84, 86, 89, 91
child, 21, 84, 88, 91, 92
Replicate/Join, 173
replicate/join composition formalism, 83, 84
composed model editor, 84, 85
Replicate/Join model, 167
reward, 1, 4, 5, 22, 23, 57, 38, 97, 97, 99, 104, 179, 186, 193, 206, 209, 211, 212, 243
accumulated reward, 193
impulse reward, 22, 99, 100, 100, 101, 164
rate reward, 193, 243
reward model
child, 97, 99, 100
INDEX

reward variable, 4, 101, 103, 180, 193, 243, 244, 247

distribution of, 4, 22, 23, 104, 157, 192, 206, 209

mean of, 202

mean of, 4, 22, 23, 104, 192, 196, 197, 199, 200, 204

not-a-number (NaN) value, 198

variance of, 4, 22, 23, 104, 182, 192, 196, 199, 200, 202, 204, 206, 209

transient, 179

run in the background, 195

run name, 161, 169, 170, 181, 182, 186, 250

SAN, 1, 2, 7, 21, 49, 50, 52–54, 56–58, 60–63, 67, 78, 83, 84, 100, 196, 197, 199, 200, 202

selected systems, 183, 250

set study, 110, 110–112

shared state variable

in symbolic state space generator, 168

simulation, 1, 2, 5, 22, 23, 37, 67, 102, 104, 157, 159, 177, 179, 188, 192, 233, 246, 247, 249, 250

on multiple machines, 180, 183, 184, 250

parameters, 177, 177, 178, 182

steady-state, see steady-state, simulation

transient, see transient simulation

type, 179, 189, 249

vs. analytic solution, 192, 194

simulator, 22, 23, 30, 34, 67, 104, 177, 177, 178, 180, 182, 189, 250

child, 249

command line options, 188

editor, 188

solver, 2, 4, 5, 7, 23, 28, 34, 37, 38, 40, 43, 45, 110, 157, 159, 161, 169, 177, 188, 191, 247, 249

child, 160, 177, 179

choosing, 192

creating, 191

solver node, 215

sparse matrix, see sparse matrix representation

sparse matrix representation, 166, 168

vs. symbolic data structure, 166

stabilizing, 160

state space, 4, 22, 23, 37, 83, 84, 159, 163, 165, 172, 174, 175, 191, 194, 195, 198, 202, 205, 235, 236, 239–242

247 249

unlumped, 84

state space generator, 30, 196

child, 160, 161

command line options, 164

symbolic, 165, 169, 174

command line options, 174

state transition matrix, 166, 168, 173

state variable, 3, 5, 7, 21, 83, 84, 88, 92, 99, 100, 164, 181, 243

shared, 83, 89, 91

detection

in adiss, 208

in diss, 205

state-level lumping, 167

state-level lumping, 167

state-sharing, 83, 84, 88, 88, 90

steady-state, 22, 102, 177, 179, 180, 189, 192, 201, 205, 208

detection

in adiss, 208

in diss, 205

initial transient, 23, 102, 103

variable, 193

stiff model

in iss, 201
INDEX

269

stochastic process, 7, 159, 164
structure
as data type, 53, 55, 60, 62
study, 22, 37, 38, 105, 105, 105, 106, 111
112, 160, 161, 163, 169, 172
177, 179, 195, 245, 247, 249
child, 105, 110

current study in simulator, 177, 249
submatrix cache size, 175
submodel, 4, 20, 21, 83, 84, 86–92, 97,
99, 235, 236, 240–243
successive over relaxation (SOR)
in diss, 204
in iss, 198
successive over relaxation solver, 168
symbolic state space generator, 167
symbolic data structure, see data structure, symbolic
symbolic state space generator, see state
space generator, symbolic
appropriate decomposition for
iterative steady state solver and
limitations of
parameters of
phases of
progress bars
usage guidelines
vs. flat state space generator
symmetry, 84, 84
optimizations, 21, 84
system time, 196

Takahashi steady-state solver (iss), see numerical solver
Tause worthe, 180
time, 101
timed transition, see Buckets and Balls,
transition, timed
tokens, 49
tools
project manager
tools menu, 35
top-level model, 97
trace level, 181
trace level, 161, 163, 165, 170, 174, 175
trace output, 30, 32, 161, 170, 181
transient simulation, 23, 177, 180, 249

UltraSAN, 1, 23, 39
unarchive, see project, unarchive
uniformization
in ars, 212
in ats, 212
in diss, 204
in trs, 210
union cache size, 175
unique table, 171
unlumped state space, 171, 173
user time, 196
user-defined functions, 45
user-defined header, 45
user-defined library, 46
vanishing state, 168
variance, 186
verbosity, 195
well-specified, 160
zero on the diagonal
in iss, 200