SIMULATION DEBUGGING AND VISUALIZATION IN THE MÖBIUS MODELING FRAMEWORK

BY

CRAIG THOMAS BUCHANAN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Adviser:

Professor William H. Sanders

ABSTRACT

Large and complex models can be difficult to analyze using static analysis results from current tools, including the Möbius modeling framework, which provides a powerful, formalism-independent, discrete-event simulator that outputs static results such as execution traces. The Möbius Simulation Debugger and Visualization (MSDV) feature adds user interaction to running simulations to provide a more transparent view into the dynamics of the models under consideration. This thesis discusses the details of the design and implementation of this feature in the Möbius modeling environment. Also, a case study is presented to demonstrate the new capabilities provided by the feature.

To my family, for their love and support

ACKNOWLEDGMENTS

I would like to thank my adviser, William H. Sanders, for his support and guidance. I would also like to thank Ken Keefe for his advice of the direction of my work with his professional insight of the Möbius tool. This work would not have been possible without their help. I would like to thank my fellow PERFORM members, especially Carmen Cheh, Uttam Thakore, David Grochocki, Ahmed Fawaz, Sobir Bazarbayev, Doug Eskins, Gabe Weaver, Ron Wright, Atul Bohara, and Robin Berthier, for their feedback, encouragement, and friendship. I would also like thank Jenny Applequist for her editorial assistance and support of this work, and previous works.

This material is based on research sponsored by the U.S. Department of Homeland Security, under agreement number HSHQDC-13-C-B0014. The U.S. government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation thereon. This material was also indirectly supported by Sandia National Laboratories. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.1.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ADVISE	ADversary VIew Security Evaluation
AFI	Abstract Functional Interface
AMI	Advanced metering infrastructure
GDB	GNU Project debugger
GUI	Graphical user interface
HITOP	Human-Influenced Task-Oriented Process
HTTP	Hypertext Transfer Protocol
IDS	Intrusion detection system
MSDV	Möbius Simulation Debugger and Visualization
OCL	Object Constraint Language
SAN	Stochastic activity network
UML	Unified Modeling Language

CHAPTER 1

INTRODUCTION

Because of its high flexibility and relative simplicity, discrete-event simulation remains a popular technique for complex analysis in many technical disciplines, as it is used in applications that range from availability assessments in computer science [1], to environmental impact assessments [2], to disease propagation assessments [3][4][5]. Despite its powerful benefits, acquisition of appropriate parameters and design of correct models of systems can be quite complicated because of the multitude of uncertainties inherent to the complex systems under study. Currently employed discrete-event simulation tools, such as Möbius [6][7], Simul8 [8], and Vensim [9], require complete models coupled with complete simulation runs to return any useful results; tweaking of model and simulation parameters can become time-consuming and error-prone, as human operators must complete each modeling workflow from beginning to end. We address that problem by introducing the Möbius Simulation Debugger and Visualization (MSDV) feature, an extension of the discrete-event simulator available in the Möbius modeling framework [10][11][12], which adds user interaction and visibility to running simulations.

The goal of the MSDV feature is to provide the analyst with a highly transparent view of the running simulation, rather than simply provide results at the end of the simulation. The transparency of both the visualization and model state modification functionalities can aid analysts in designing correct, complete models of the complex systems under consideration. The additional functionality effectively increases the ease, speed, and reliability of the model validation and verification phases of the overall simulation analysis.

In this chapter, we present an overview of the Möbius modeling framework, we discuss the current state of the Möbius discrete-event simulator including its user interface and its visualization of results, and we more clearly define the motivation and goals of the project. In the following chapters, we describe the specific functionality provided by the MSDV feature (Chapter 2), we examine how it is implemented in the Möbius modeling framework (Chapter 3), we consider a case study of an attack on an advanced metering infrastructure (AMI) network to reveal the utility of the new features (Chapter 4), and we conclude by speculating about the future direction of this work (Chapter 5).

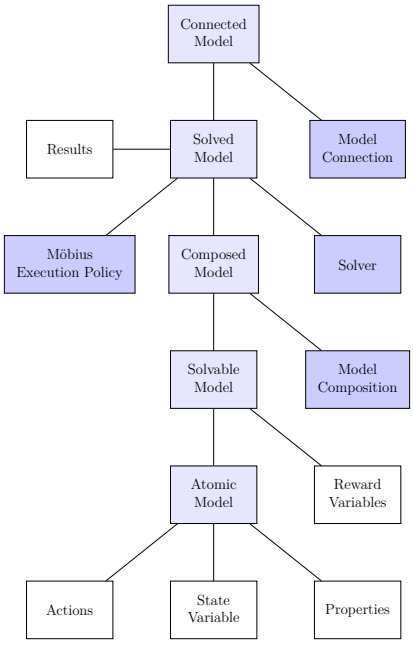


Figure 1.1: Möbius Framework Components.

1.1 Möbius Overview

Möbius is an extensible, model-based framework used to model stochastic, discrete-event systems. Its extensibility is due to its support of multiple domain-specific modeling formalisms, all of which are inherited from its rich, base modeling formalism. This base modeling formalism, referred to as the Abstract Functional Interface (AFI) [13][14][15], allows the development and support of new domain-specific modeling formalisms in the Möbius modeling framework. The flexibility of being able to define new domain-specific modeling formalisms in Möbius is one of its most valuable features, since new formalisms can always be invented to address the continuous stream of new systems that must be investigated.

A fully specified Möbius model, or *solvable* model, is a set composed of several Möbius components: atomic model(s), composed model(s) (optional), a reward model, a study, and a solver, as shown in Figure 1.1 [16].

An atomic model is the most basic model, describing a component of the system under consideration. An atomic model is defined using a single, specific atomic modeling formalism, such as a stochastic activity network (SAN) [17][18][19], a fault tree, a buckets-and-balls model, an ADversary VIew Security Evaluation (ADVISE) model [20][21], or a Human-Influenced Task-Oriented Process (HITOP) model [22]. All of those modeling formalisms are inherited directly from the AFI base classes, allowing Möbius to internally represent all relevant information in the AFI form. The AFI representation of the models allows

compatibility between the different atomic modeling formalisms, while allowing them to utilize the same solution technique libraries.

A composed model is a higher-level model that describes how the components of the system are interrelated. The Rep/Join model, one of the most commonly used composed models, combines specified state variables of the atomic submodels into single shared state variables, effectively allowing the atomic submodels to communicate. Composed models allow users to create large models that are manageable instead of enormous, unwieldy, internally-repetitive atomic models.

A reward model allows certain attributes of the model to be specified as observable during the model-solving time [23]. A study defines the values of the *global variables*[1] in the model.

A solver specifies the solution technique and parameters to solve the model. The Möbius modeling framework supports several solution techniques, including both discrete-event simulators and numerical solvers that are based on both transient and steady-state algorithms. The purpose of the solver is to calculate the model attributes defined in the reward model using the specified solution technique and parameters.

End users specify Möbius solvable models using the Java-based Möbius GUI. Möbius then converts those XML specifications into C++ runnable models. The runnable models are compiled with the Möbius back-end libraries to form complete binary executables to solve the originally specified solvable models. The binary executables relay status messages and results to the Java-based Möbius GUI through the Möbius communication layer[2] to be displayed in human-readable form.

1.2 Möbius Discrete-Event Simulator Overview

Each solver in the Möbius modeling framework, including the Möbius discrete-event simulator, executes modeling-formalism-independent solution techniques by decoupling the solution technique used from the specific modeling formalism of the model under consideration. That powerful feature makes it possible to solve a large subset of modeling formalisms, as well as to easily combine submodels created in different modeling formalisms within this subset. To accomplish such independence in the solution technique, the Möbius modeling framework

[1] A *global variable* in Möbius is a variable defined outside of a specific model. This functionality allows a modeler to define a variable across multiple models, and quickly modify it to solve a series of experiments with different values.

[2] The Möbius communication layer provides the communication medium between the back-end Möbius simulation processes and the front-end Möbius user interface. It operates by forwarding TCP/IP messages between the POSIX sockets of the two end-layers. Its pertinence to the MSDV feature is discussed in Section 3.2.

utilizes the Abstract Functional Interface (AFI), a general modeling formalism that leverages the two overarching modeling characteristics shared by many modeling formalisms: the model state and the transition system [13].

In the AFI, a *state variable* is a basic modeling element that represents the state of a component within the model [16]. For example, when a queue is being modeled, a state variable can represent the number of items currently in the queue. Then, the full model state can be represented as the set of all state variables' values.

Also, in the AFI, an *action* is a basic modeling element that changes the model state [16]. Each action is associated with a timing distribution (e.g., exponential or Weibull) that determines when it will fire, thus changing the model state. Each action is also associated with a Boolean "enabled" status to determine whether it is currently eligible to fire. That status is determined by certain specified conditions of the model state. For example, in the queue model mentioned in the previous paragraph, an action can represent the removal of an item from the front of the queue. The action could be defined as exponential with a rate of 1.0, where it is only enabled when there is at least one item in the queue. Therefore, the full transition system of the model can be represented as the set of all actions in the model.

In addition to representing the model, the Möbius discrete-event simulator also employs a *future event list* to determine the specific sequence of *events* in the given simulation batch [10]. In the list, each event couples an action with a deterministic simulation time at which it will fire. The list contains one event item per enabled action. For example, in the same model, the action is sampled at the initialization of the simulation, which could result in a value of 0.95 (since the timing distribution is not deterministic). At this time, the action is inserted into the future events list with the deterministic time of 0.95. Next, at simulation time $t = 0.95$, the action is fired and the associated event is removed from the event list. Assuming the action is still enabled, or there are more people in the queue, the action will be sampled again, which could result in a value of 1.15. At this time, the action is inserted into the future events list with a deterministic time of 2.10 (current simulation time + sampled distribution time). At simulation time $t = 2.10$, the action is fired and the associated event is, once again, removed from the future events list.

The general algorithm for the Möbius simulator, as originally presented in [10], is shown in Figure 1.2 [10]. In this algorithm, E represents the future event list, μ represents the model state in terms of the culmination of the values of all of the state variables in the model, EN_μ represents the set of actions that are enabled in the model state μ, and e_a represents the event associated with action a. First, the algorithm generates the initial future event list E by adding an event for each action that is enabled in the initial model state. Next, the earliest event in the future event list E is fired. After reaching this new model state μ,

$E = 0$
$\mu =$ `INITIAL MARKING`
$\forall a \in EN_\mu$
    $e_a = GenerateEvent(a, \mu)$
    $E = E \cup \{e_a\}$
$while(E \neq 0)$
    $e_a = Earliest(E)$
    $E = E - \{e_a\}$
    $\mu' = FireEvent(a, \mu)$
    $\forall e_a \in E$
        $if(a \notin EN_{\mu'})$
            $E = E - \{e_a\}$
    $\forall a \in EN_{\mu'}$
        $if(a \notin E)$
            $e_a = GenerateEvent(a, \mu')$
            $E = E \cup \{e_a\}$
end

Figure 1.2: Möbius Discrete-Event Simulation Algorithm.

the future event list E is updated by removing events associated with actions that are not enabled, and adding events associated with actions that have become enabled. This process continues until one of two conditions has been met. The first case is when the model reaches an absorbing state, a state in which the future event list is empty. This case marks the end of a simulation batch since, in this state, no actions are enabled to change the model to another state. The second case is when the maximum simulation run time has been reached. In Möbius, this case typically occurs at the end of the accumulation period for all specified reward variables, since, traditionally, these are the only observable values in the model.

1.3 Simulation Results in Möbius

As described in Section 1.1, the goal of solvers is to evaluate the model attributes defined in the associated reward model. The Möbius discrete-event simulator, for example, runs a specified number of full simulations, each referred to as a *batch*. After each set of batches, the back-end C++ simulator executable processes relay status and result information back to the front-end Java-based GUI. As described in [24], those data are stored in the Results Database, which allows the user to query model information such as submodel names, model versions, date, analysis technique, and model parameter values. Those data could then be used either to create plots of simulation data of interest, or to output data to an

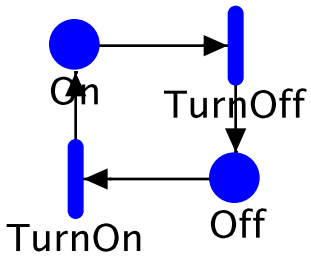


Figure 1.3: Simple SAN Model (On/Off).

external trace file to be interpreted by a third-party visualization tool, such as Traviando [25][26][27][28][29][30].

To examine the workflow of creating and analyzing simulation result data in Möbius, consider the simple SAN model presented in Figure 1.3. In it, a single token traverses the model from the `On` place to the `Off` place and back again through the exponentially distributed activities `TurnOff` and `TurnOn`, both of which have a rate of 1.0. To get a full view of the model after the simulation, we define the reward model to contain two variables, one containing the value of `ON` and another containing the value of `Off`. We also set each variable to be evaluated on the simulation time interval $t = [0.0, 10.0]$. Next, we define a default range study and simulator. On running the simulator with full trace output, we obtain a text trace and the results of the simulation, which are shown in Figures 1.4 and 1.5, respectively.

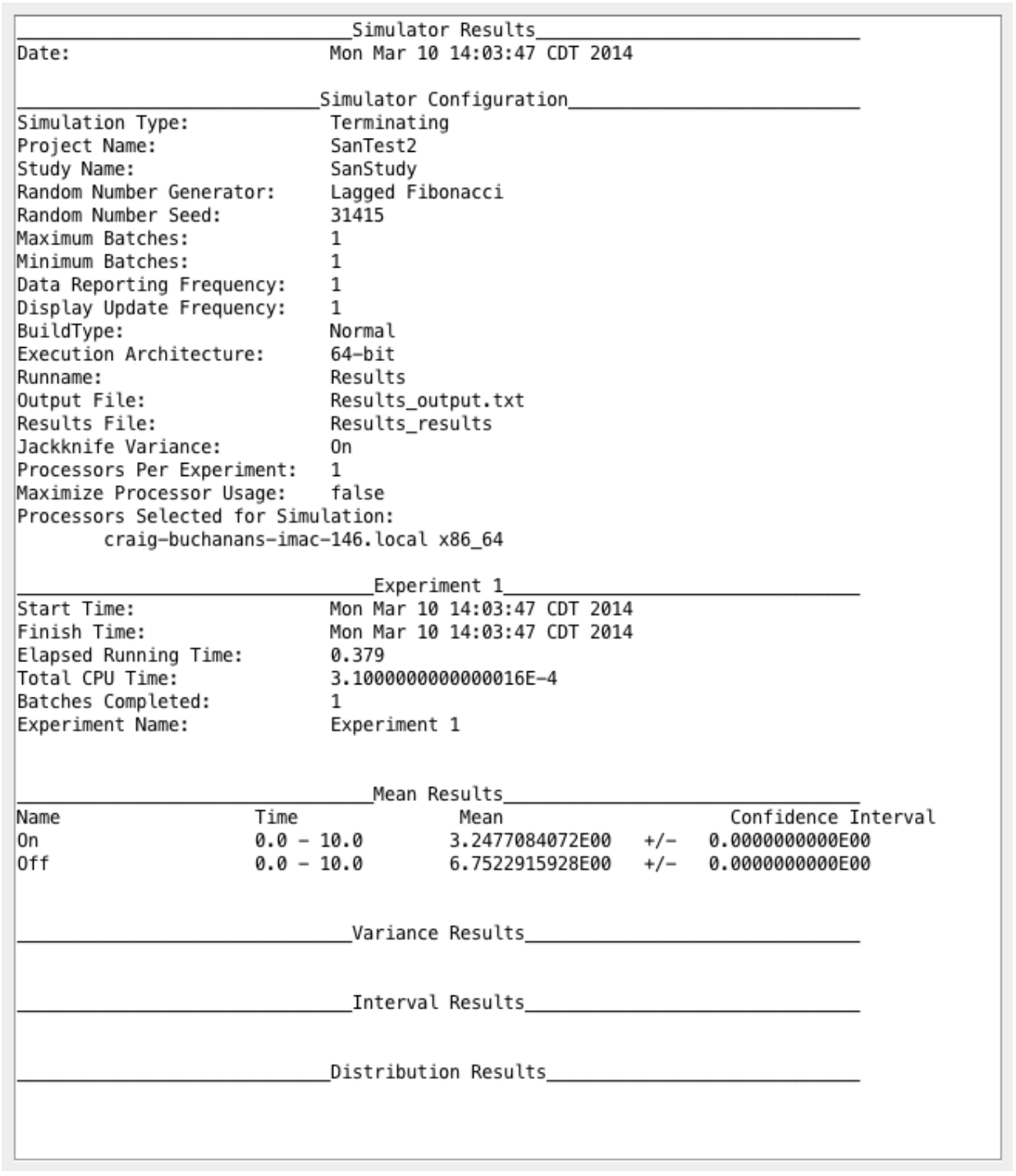
The full simulation trace contains all of the model states between firing actions, details about each firing action, the enabling state of each action before it fires, and the affected state of each action after it fires (discussed in Section 2.2). From that information, third-party tools, such as Traviando [25], can effectively generate visualizations of the terminated simulation batch.

The simulation results page summarizes the results of the entire set of simulation batches of the model as specified in the associated reward model. In our example, the two variables of interest are the `On` and `Off` values during the simulation time interval $t = [0.0, 10.0]$. Those values can be observed in the "Mean Results" section of the results page (see Figure 1.5). As expected, we can see that the single token resided in each of the places for approximately half of the simulation time (`On->Mark()` $\approx 32.5\%$, `Off->Mark()` $\approx 67.5\%$). Note that the values are more likely to converge to a value closer to the ideal if a larger set of simulation batches is executed. Also, note that in this example, the confidence interval of both variables is zero, because only a single simulation batch was executed.

Although we gain valuable insight into models using current simulation result visualization

```
************************************
                TRACE
************************************
##### NEW BATCH #####
Initial Model State:
  Model: (2)SanModel
    On = 1
    Off = 0
Enabling state before action firing:
  On = 1
#### Firing: 0.0159023  Action: (2)SanModel->TurnOff
#### After: (2)TurnOff->TurnOff
Affected state after firing:
  On = 0
  Off = 1
Model State:
  Model: (2)SanModel
    On = 0
    Off = 1
Enabling state before action firing:
  Off = 1
#### Firing: 0.255223  Action: (2)SanModel->TurnOn
#### After: (2)TurnOn->TurnOn
Affected state after firing:
  Off = 0
  On = 1
Model State:
  Model: (2)SanModel
    On = 1
    Off = 0
```

Figure 1.4: SAN Model (On/Off) Simulation Trace Snippet.



```
_____________________________Simulator Results_____________________________
Date:                          Mon Mar 10 14:03:47 CDT 2014

_____________________________Simulator Configuration_____________________________
Simulation Type:               Terminating
Project Name:                  SanTest2
Study Name:                    SanStudy
Random Number Generator:       Lagged Fibonacci
Random Number Seed:            31415
Maximum Batches:               1
Minimum Batches:               1
Data Reporting Frequency:      1
Display Update Frequency:      1
BuildType:                     Normal
Execution Architecture:        64-bit
Runname:                       Results
Output File:                   Results_output.txt
Results File:                  Results_results
Jackknife Variance:            On
Processors Per Experiment:     1
Maximize Processor Usage:      false
Processors Selected for Simulation:
        craig-buchanans-imac-146.local x86_64

_____________________________Experiment 1_____________________________
Start Time:                    Mon Mar 10 14:03:47 CDT 2014
Finish Time:                   Mon Mar 10 14:03:47 CDT 2014
Elapsed Running Time:          0.379
Total CPU Time:                3.1000000000000016E-4
Batches Completed:             1
Experiment Name:               Experiment 1


_____________________________Mean Results_____________________________
Name                  Time              Mean                      Confidence Interval
On                    0.0 - 10.0        3.2477084072E00   +/-   0.0000000000E00
Off                   0.0 - 10.0        6.7522915928E00   +/-   0.0000000000E00


_____________________________Variance Results_____________________________


_____________________________Interval Results_____________________________


_____________________________Distribution Results_____________________________
```

Figure 1.5: SAN Model (On/Off) Simulation Results.

tools, their current implementations impose unnecessary analysis restrictions that we could work to overcome. Those restrictions are discussed next.

1.4 Motivation

Although each set of simulation batch results, as discussed in the previous section, provides a multitude of analytic data, the acquisition of such data is hindered by the restriction that the simulation must be terminated. In other words, the user does not have access to any of the analytical data until the entire simulation batch is complete. For long or complex simulations, that could not only take a long time, but also result in an output trace so enormous that it may be virtually useless to the simulation analyst. Also, if there is a bug in the simulation, the simulation may never terminate, or may terminate before the batch is done and the results can be reported. Those bug-related issues could obfuscate data that could lead the analyst to a more precise location of the error in question.

Another limitation imposed by the simulation termination restriction is the inability to modify the model state during the running simulation. Although such additional simulation interactivity may not be necessary for complete models, it could greatly increase the speed and ease of the workflow of tweaking the model parameters to find the appropriate model for the system under consideration. This additional simulation-interaction functionality is in contrast with the existing approach of updating the initial model parameters only in the model-creation stage, and then working all the way through to the final stages of running complete simulations of the entire model.

The main motivation for this project is to address those simulation restrictions to achieve a higher overall credibility of models and their respective simulations. To increase overall credibility, the model and its simulations, in their entirety, must be more easily understood by both the modeler and third-party viewers. The third-party viewers may include, but are not limited to, project members such as clients, managers, and students. To achieve the desired level of credibility, the model must be both validated and verified.

Model validation in the Möbius modeling framework consists of two parts. First, it ensures that the model specification conforms to the specified modeling formalism. That part can be automated through formal definition of each modeling formalism implementation in Möbius. The process is described for several specific modeling formalisms in Section 3.3. Second, model validation in Möbius also ensures that the model correctly represents the system under consideration. In other words, it ensures that the model satisfies all of the specified project requirements of the system analysis. This part is more difficult to automate, since

it is project-specific and often somewhat qualitative in nature.

Model verification, in contrast to validation, ensures that the model is correctly executing as expected. In other words, it ensures that the validated model actually works. It, too, is difficult to automate, as it, too, is project-specific and often somewhat qualitative in nature.

In order to accomplish both model validation and verification in the Möbius modeling framework, the modeler and the third-party viewers must be able to easily and fully understand as much of the model as possible. For maximum model credibility, it is crucial that this understanding occurs in both the model creation and model analysis stages of the project.

1.4.1 Model Creation

In order to create a useful model, the modeler must fully understand the details of the model under construction. Some things that often prevent the modeler from having that understanding include model complexity, inadequate understanding of a constituent modeling formalism, and simple human error.

As models grow to encompass more precise representations of systems under consideration, they quickly become immensely more complex. While that additional complexity is true for single atomic models, it is even more true for composed models that comprise multiple atomic and composed models, and potentially incorporate multiple modeling formalisms. That inevitable increase in overall model complexity makes model creation more cumbersome, which potentially limits the practically feasible size of some models for less experienced users.

Also, in addition to the multitude of existing modeling formalisms, new modeling formalisms are continually being developed to address new problem cases more effectively. Given the large and growing set of available modeling formalisms, it is a mistake to assume that every modeler will be entirely familiar with every one. For example, a modeler may choose to use the basic set of model elements within the SAN modeling formalism without complete knowledge of the entire formalism. In that case, a seemingly trivial detail about the modeling formalism, such as the time at which the input predicate of an activity executes, may be ignorantly ignored by the modeler. If such behavior results in unexpected behavior in the model, finding the root of the issue may be difficult with the current simulation functionality in Möbius.

Beyond those specific issues, there is the general problem that the modelers under consideration are human. For that reason, they are likely to make mistakes that could lead to

unexpected model behavior. The problem is exacerbated when a project requires collaboration among multiple people or groups of people, as is often the case. With the current simulation functionality in Möbius, it is difficult for another user to quickly understand the functional details of a complicated model and its simulation.

1.4.2 Model Analysis

To successfully analyze a model, a user must fully understand the relevant functional subset of the model under consideration. Issues that may prevent the user from achieving that understanding include model complexity and scalability issues of the visualization. The issue of model complexity is discussed above (Section 1.4.1) in the context of model creation. The scalability of simulation visualizations, however, has not yet been thoroughly examined. Several tools for post-simulation analysis, such as Traviando [25], create visualizations that show as much information as possible from generated simulation trace results. However, since those tools are restricted to post-simulation analysis, they cannot access data during the running simulation. Also, since those tools show a great amount of detail about previously executed simulations, they typically do not scale well, which causes them to be less useful on larger and more complex models.

1.5 Goals

The primary goal of this project is to address the previously discussed interaction and visualization limitations imposed by the current state of the Möbius discrete-event simulator by introducing the Möbius Simulation Debugging and Visualization (MSDV) feature. This feature will address the limitations by extending the Möbius discrete-event simulator to provide the additional features discussed in Chapter 2. In addition, MSDV is designed in the form of adaptable modules to facilitate the development of extensions for specific modeling formalisms. Another goal of this project is to fully develop extensions for several existing modeling formalisms, both atomic and composed. Those secondary design and implementation goals are discussed next.

1.5.1 Adaptable Modules

Modularity is an important requirement of the MSDV feature design, because Möbius is intended to be highly extensible. Specifically, the MSDV feature must be able to easily

extend its current functionality to a multitude of specific modeling formalisms, including ones that have yet to be developed. The reason is that some parts of the MSDV feature, such as the visualization aspect, are modeling-formalism-specific. For example, the visualization of the data for a SAN model simulation should be composed only of SAN modeling elements, such as places, activities, input gates, output gates, and arcs. Similarly, the visualization of the data for a Rep/Join model should be composed only of Rep/Join model elements. Since the visualization of the different modeling formalisms must be composed of modeling-formalism-specific elements, these portions of the feature must be developed independently from one another. The goal of the modular design in the MSDV feature development is to minimize the extent of independently developed portions of the feature while maximizing the reusable code that is modeling-formalism-independent.

To reach the goal of maximum adaptability, the MSDV feature will contain a base module that leverages the AFI [13]. The AFI is a generic atomic modeling formalism from which all other modeling formalisms in Möbius, both atomic and composed, are derived. When a solver in the Möbius modeling framework, including the Möbius discrete-event simulator, is being run, the specified model is compiled as an AFI model regardless of its original modeling formalism. Since all models in the Möbius modeling framework are thereby guaranteed to be representable in the low-level AFI modeling formalism, the base AFI module in the MSDV feature effectively covers every modeling formalism, both present and future. From that starting point, it is relatively simple for a developer to build modeling-formalism-specific visualization extensions for the MSDV feature, and the developer can focus on the modeling-formalism-specific details of the extension, as the lower-level functionality of the feature is effectively encapsulated.

1.5.2 Atomic Model Implementation

After creating the AFI base module of the MSDV feature, the next goal is to develop modeling-formalism-specific modules for each atomic modeling formalism that is currently implemented in Möbius. The goal for each of these specific modeling formalisms is to represent the data to the end user using only the modeling elements available in that specific formalism. For example, if the user is running a simulation on a model specified using the SAN modeling formalism, then the resulting visualization will be represented with SAN-specific modeling elements, such as places, activities, input gates, output gates, and arcs. The two specific atomic modeling formalisms that will be examined in this thesis are stochastic activity networks (SAN) and ADversary VIew Security Evaluation (ADVISE) models.

The details of each end-user visualization are discussed in Section 3.3.2.

1.5.3 Composed Model Implementation

In addition to implementation of several atomic modeling formalisms, another goal of the project described in this thesis is to implement a visualization extension for the commonly used Rep/Join composed modeling formalism. This state-sharing composed modeling formalism is inherently more complex than atomic modeling formalisms, since it is composed of multiple atomic modeling formalisms. Because of its increased complexity, its visualization, too, must be more complex. A major goal is to retain scalability as effectively as possible, since the number of states can quickly multiply with a few Replication elements. The details of the specific implementation are discussed in Section 3.3.3.

CHAPTER 2

FEATURES

In order to achieve the desired transparency and usability, we first had to consider the additional interface features needed for the discrete-event simulator. We needed a way to access the model state during the simulation, a way to modify the model state during the simulation, a way to pause the progress of the running simulation, and a way for users to interact with the simulation. For the first requirement, we implemented the functionality of model state analysis. For the second, we implemented a reliable way to effectively modify the model state. For the third, we implemented a way to apply explicitly defined breakpoints to the running simulation and a way to implicitly step through the simulation. Finally, for the last requirement, we implemented a graphical user interface (GUI) for the model that provides access to all those new features. The new features are discussed throughout this chapter.

Also, to explore each of those features in a more tangible manner, we will examine them in terms of a running simulation batch associated with the simple AFI model presented in Figure 2.1. The model is composed of three state variables (`SV1`, `SV2`, `SV3`) and three actions (`A1`, `A2`, `A3`). The initial state s_0 of the model is as follows:

```
SV1->Mark() = 2
SV2->Mark() = 0
SV3->Mark() = 0

A1->Distribution(Deterministic, 1.00)
A2->Distribution(Deterministic, 0.50)
A3->Distribution(Deterministic, 1.75)
```

2.1 Model State Analysis

In the Möbius modeling framework, the model state is composed of both the culmination of the values of the state variables [16] and the contents of the future event list [10]. The con-

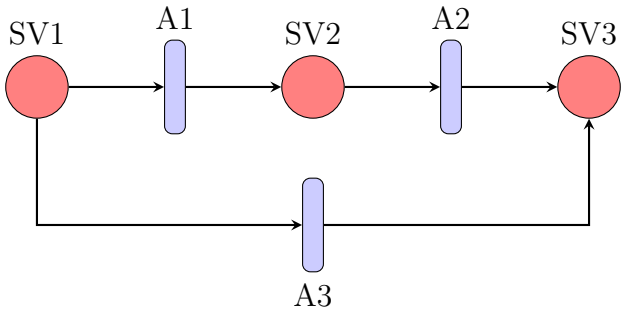


Figure 2.1: AFI Simple Flow Example.

tents of the state variables are stored in a contiguous memory block, and the contents of the future event list can be accessed in a straightforward manner. The values must be serialized into a message and sent to the Möbius visualization front-end over the communication layer, as discussed in Section 3.2.

Returning to the running AFI example model presented at the beginning of this chapter, we can represent the original model state s_0 as the following:

```
SV1->Mark() = 2
SV2->Mark() = 0
SV3->Mark() = 0

FEL = { {A1, 1.00}, {A3, 1.75} }
```

where the `Mark()` method represents the value of the state variable, and `FEL` represents the future event list in the form $\{E_0, \ldots, E_n\}$, where E represents an event in the form $\{A, T\}$, where A represents the action associated with the event, and T represents the simulation time t at which the event is to fire.

Continuing the simulation to simulation time $t = 1.25$, after action `A1` has fired, we can examine the resulting model state:

```
SV1->Mark() = 1
SV2->Mark() = 1
SV3->Mark() = 0

FEL = { {A2, 1.50}, {A3, 1.75}, {A1, 2.00} }
```

All of the data will be represented in the MSDV module for AFI models. For modeling formalisms that have been explicitly defined in the MSDV feature, the data will be converted

into modeling elements belonging to that specific modeling formalism in the form of a visualization interface. For modeling formalisms that have not been explicitly defined in the MSDV feature, the data will by default be represented with modeling elements belonging to the AFI visualization interface. The specific functionality is discussed in Section 2.5.

2.2 Model State Modification

In addition to displaying the model state at a given moment in the simulation, the MSDV feature also provides a means to modify the model state at that given moment. That functionality allows the user to more thoroughly examine a simulation from that specific time, which could be useful if the modeler is debugging a model or would like to examine what other possible behaviors the simulation could exhibit without having to change the actual model. For example, the user could modify the model state to force rare events to occur to check difficult model behavior, such as event interleaving.

As discussed in Section 2.1, the model state of an AFI model is represented through the values of the state variables and the contents of the event list. Thus, the model state can be fully modified if those two aspects can be modified. Therefore, the MSDV feature offers the capability of modifying the values of the state variables and the contents of the event list.

Model state modification is more complicated than model state analysis in the Möbius modeling framework. Its added complexity is a result of the dependencies between the elements of the model state. For example, modifying the value of a single state variable could result in a change to the enabling status of an action, thus affecting the contents of the future event list. To address such dependencies, we use the built-in dependency mechanisms of the Möbius modeling framework.

Those dependency mechanisms, as presented in [10], operate by associating state variables with actions by declaring the state variables to be either `enabling` or `affecting` with respect to the actions. If a state variable is marked as `enabling` to an action, then modifying that state variable would require that action to reevaluate its enabled status. If an action is marked as `affecting` a state variable, then when that action fires, the state variable value may be altered by the firing event.

For example, consider the simple AFI model, derived from [10], that is pictured in Figure 2.2. The model shows the `enabled` and `affected` relationships between the state variables and actions. As can be seen, an `enabling` relationship exists between state variable `P2` and action `A2`, since the enabled status of `A2` depends on the value of `P2`. Also, an `affected` relationship exists between the action `A2` and the two state variables `P2` and `P3`, since the

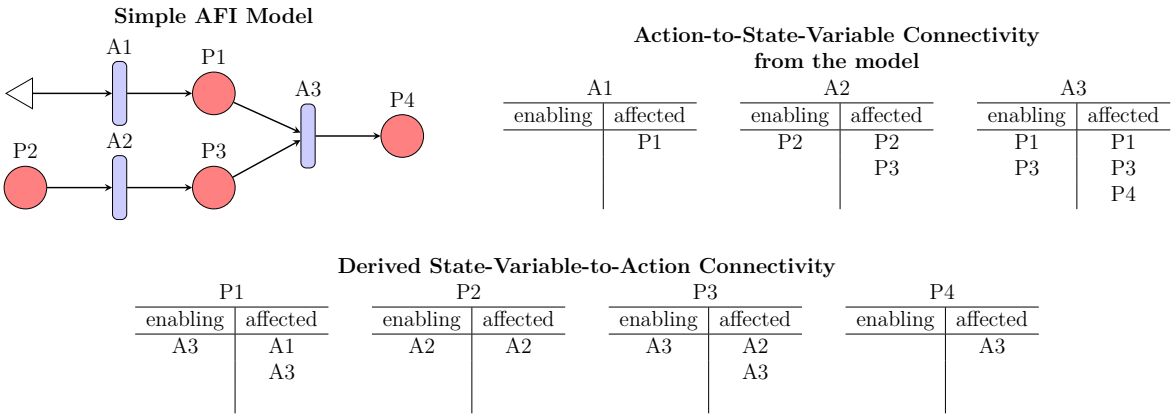


Figure 2.2: Connectivity List Example.

firing of `A2` could result in changes in the values of both state variables. Use of those relationships simplifies the modification of the model state, since only `affected` state variables must be reevaluated.

Since the model state modifications available through MSDV only include modification of state variables and the firing times of the events in the future events list, the only model state relationships we need to examine are the `enabling` relationships. The reason is that our model state modifications cannot directly alter actions.

If we continue the AFI simulation example from Section 2.1, the model state at simulation time $t = 1.25$ can be represented as follows:

```
SV1->Mark() = 1
SV2->Mark() = 1
SV3->Mark() = 0

FEL = { {A2, 1.50}, {A3, 1.75}, {A1, 2.00} }
```

Altering the model state by removing one token from `SV1` and adding one token to `SV2` changes the model state to the following:

```
SV1->Mark() = 0
SV2->Mark() = 2
SV3->Mark() = 0

FEL = { {A2, 1.50} }
```

As seen in the modified model state, altering the values of the state variables can also affect the contents of the future event list. The reason is that removal of the final token from

SV1 disables the actions A1 and A3, and hence removes them from the future event list. Only action A2 remains in the event list, since it is still enabled with at least one token in SV2. Continuing the simulation from simulation time $t = 1.25$ to $t = 1.75$ results in the following model state:

```
SV1->Mark() = 0
SV2->Mark() = 1
SV3->Mark() = 1

FEL = { {A2, 2.00} }
```

Between the previous two model states, action A2 fires, moving a token from SV2 to SV3. Action A2 is still enabled, since at least one token remains in SV2.

In addition to modifying the values of state variables, the MSDV feature also allows users to directly modify the firing times of the events in the future event list to any times in the simulation time interval $t \geq$ currentSimulationTime. That modification can be done regardless of the timing distribution of the action associated with the specific event, since in Möbius the timing of each event becomes deterministic when it is added to the future event list. For example, to modify the original model state at simulation time $t = 1.25$, the user can modify the firing time of the event associated with A1 from $T = 2.00$ to $T = 1.35$ to ensure that it fires before the other two actions. After that change has been made, the current model state at simulation time $t = 1.25$ becomes:

```
SV1->Mark() = 1
SV2->Mark() = 1
SV3->Mark() = 0

FEL = { {A1, 1.35}, {A2, 1.50}, {A3, 1.75} }
```

Running the simulation for 0.15 time units to $t = 1.40$ results in the following model state:

```
SV1->Mark() = 0
SV2->Mark() = 2
SV3->Mark() = 0

FEL = { {A2, 1.50}, {A3, 1.75} }
```

2.3 Breakpoints

To access or modify the model state, the user must have a way to pause a running simulation. One way to pause it is through user-defined simulation breakpoints, which allow the user to explicitly define conditions under which the running set of simulation batches should pause. There are three types of breakpoints: simulation time breakpoints, action breakpoints, and state variable breakpoints. Each type returns a `Boolean` value, allowing the user to easily create combinations of the three types using the Boolean logical operators `AND`, `OR`, and `NOT`.

Note that simulation breakpoints are not limited to a single simulation batch. For example, consider an action `A4` that is so rare that it fires only once, in the 9215th batch of the simulation. Setting an action breakpoint (discussed in Section 2.3.2) on this action `A4` would pause the simulation on the 9215th batch after continuing through the previous batches.

2.3.1 Simulation Time Breakpoints

A simulation time breakpoint allows the user to pause the simulation at a certain simulation time t. For example, if the user sets the simulation to run until simulation time $t = 1.25$, as in the examples in Sections 2.1 and 2.2, the simulation would pause at simulation time $t = 1.25$, allowing the user to access and modify the resulting model state at this simulation time. Note that the simulation time t does not have to coincide with any events in the future event list. However, if it does, the simulation will pause immediately following all of the events at that given simulation time t. In other words, this simulation example will pause at simulation time $t = 1.25 + \lim_{\delta t \to 0} \delta t$.

2.3.2 Action Breakpoint

An action breakpoint allows the user to pause the simulation at a certain action event of a specific action. Currently, available action events include `OnFired`, `OnStatusToEnabled`, and `OnStatusToDisabled`. Respectively, the breakpoints are triggers at the simulation times immediately following firing of actions, switching of an action from disabled status to enabled status, and switching of an action from enabled status to disabled status. The action breakpoint can be used if the user wants to run a simulation until a certain event has fired, which can be useful during examination of actions that rarely fire.

In the running example, a user could define a breakpoint when the action `A2` becomes enabled. If the simulation is started from its initial state s_0, the simulation would run until simulation time $t = 1.00$, at which point a token would have moved from state variable `SV1`

to state variable `SV2`, enabling action `A2`. In that paused state, the user would have access to the previously discussed model state analysis and model state modification functionality.

2.3.3 State Variable Breakpoint

A state variable breakpoint allows a user to pause the simulation when certain conditions concerning state variable values have been met. Specifically, arithmetic combinations of state variable values and literal values are compared using standard comparison operators: $\{<, >, =\}$. For example, if a user wants to pause a simulation when a certain state variable value is greater than another state variable value by 7.5 or more, the user can specify the breakpoint $sv1 > sv2 + 7.5$. This functionality is useful during the examination of the quantitative relationships between state variables.

More specifically, the comparison operations only use state variable primitive values, rather than simply the values of the state variables. The reason is that in the Möbius modeling framework, state variable values can be represented as complex, user-defined data structures composed of nested `struct`s and `array`s. Since these data structures can be complicated and hard to compare intuitively, our implementation restricts the comparisons to the primitive data types of which these data structures are composed. The details of accessing state variable primitive values, including the meaning of state variable indices in the MSDV feature, are discussed in Section 3.2.2.

In the running example, a user may want to analyze the overall flow of the system by pausing the simulation when the final state variable `SV3` has a larger value than the initial state variable `SV1`. In that case, the user would specify the breakpoint $sv[2][0] > sv[0][0]$, assuming that the unique state variable index value of 0 represents `SV1` and 2 represents `SV3`. As seen in Section 2.4, with no model state modification, that breakpoint will occur at simulation time $t = 1.75$ when `SV1->Mark()` $= 0$ and `SV3->Mark()` $= 2$.

2.4 Simulation Stepping

In addition to breakpoints, the MSDV feature provides simulation stepping as another means of pausing a running simulation. Whereas breakpoints are explicitly defined by users, simulation stepping is an implicitly defined operation that runs the simulation until the next action fires. Since the model state of a discrete-event simulation does not change until an action is fired, simulation stepping gives users a way to easily examine all of the successive model states of a running simulation in chronological order. The examination can occur

from any given paused state, including the initial model state, a state reached through the use of breakpoints, or a state reached through previous stepping. This functionality is useful for examining the fine-grained details of the operation of a running simulation from a given simulation time t.

Since the running example is very small, we could easily step through the entire simulation. One step from the initial state s_0 would result in the following state at simulation time $t = 1.0$:

```
SV1->Mark() = 1
SV2->Mark() = 1
SV3->Mark() = 0

FEL = { {A2, 1.50}, {A3, 1.75}, {A1, 2.50} }
```

That would be the state immediately following the firing of the action associated with the first event in the future event list, `A1`. Stepping again results in the next simulation state at simulation time $t = 1.50$ immediately after `A2` fires:

```
SV1->Mark() = 1
SV2->Mark() = 0
SV3->Mark() = 1

FEL = { {A3, 1.75}, {A1, 2.50} }
```

The next simulation step results in the final model state at simulation time $t = 1.75$ immediately after `A3` fires:

```
SV1->Mark() = 0
SV2->Mark() = 0
SV3->Mark() = 2

FEL = { }
```

That is the final model state of the simulation, since the future event list is empty. Since no more actions will fire, the model state will not change from this final state. At this point, the simulation has terminated.

2.5 Model State Visualization

To use the previously discussed features, the user requires a powerful and intuitive interface to control and view the running simulation. After considering potential designs for this interface, we decided that the most useful interface would be one already familiar to the user. Thus, we implemented the user interface to mimic the specific modeling formalism with which the user had specified the model. For example, if the initial model is defined as a stochastic activity network (SAN) model [17], then the user interface should display a SAN-like presentation of the model. Specifically, the user interface will display the model state as a combination of SAN elements similar to the SAN elements of the original model. Thus, the visualization interface becomes an effortless way to bring the user's model to life, rather than a complicated and unfamiliar tool that the user must painstakingly learn.

Although that design decision simplifies the use of the tool, it would be impractical to create a different user interface for every different modeling formalism, not only because of the large number of existing formalisms, but also because of the constant introduction of new modeling formalisms. To address the issue, we leveraged the underlying Abstract Functional Interface (AFI) of the Möbius modeling framework [13]. The model-level Möbius AFI is a modeling formalism that is the basis of all other modeling formalisms in Möbius. Since all of the specific modeling formalisms are forms of their parent AFI modeling formalism, each can be represented as an AFI model. Therefore, we started by implementing the user interface in AFI. We then continued to develop user interfaces for specific modeling formalisms. The idea is that if the user interface for a specific modeling formalism has not yet been implemented (e.g., a newly developed modeling formalism is being used), then the MSDV tool will default to the AFI visualization and user interface. Although the general AFI visualization and user interface will not be as familiar to a user as a modeling-formalism-specific version would be, it still provides the same power as formalism-specific visualizations and editing interfaces in MSDV. The details of the visualization interfaces for the supported modeling formalisms are given in Section 3.3.

CHAPTER 3

IMPLEMENTATION

The implementation of the MSDV feature in the Möbius modeling framework relies on its integration into the currently existing discrete-event simulator, which is composed of three different layers [10], pictured in Figure 3.1. They are:

Back-end Möbius Simulation Processes

Implemented in C++, this layer executes the actual simulation, and thus leverages the power and speed available from running natively on the host machine.

Communication Layer

This layer provides the medium for the communication between the back-end Möbius simulation processes and the front-end visualization interface.

Front-end Visualization Interface

Implemented in Java, this layer allows users to control and receive feedback from the back-end Möbius simulation processes.

The implementation of the MSDV feature with respect to those three simulator layers is described in this chapter.

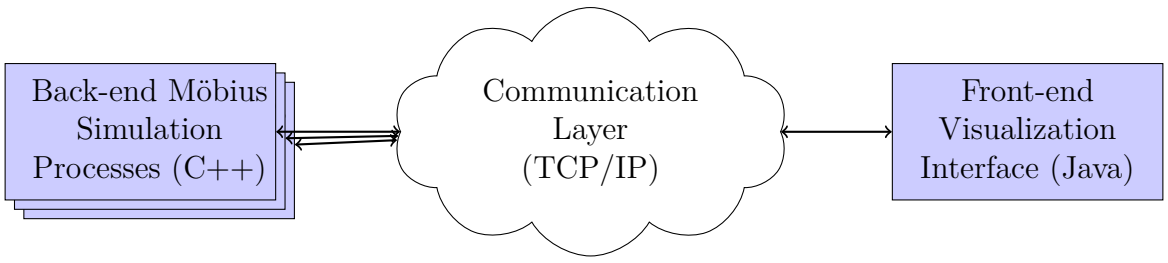


Figure 3.1: Möbius Layer Interaction.

3.1 Back-end Möbius Simulation

The back-end Möbius simulation processes are responsible for executing the actual simulation of the model. This layer must be modified to allow model state analysis, model state modification, and simulation pausing through the use of breakpoints and stepping functionality.

3.1.1 Model State Analysis

Although model analysis in Möbius typically relies on definitions of the reward variables in the reward model component of the Möbius model, the MSDV feature aims to analyze the model as a whole. Therefore, the instantaneous values of the state variables are also accessible. In the Möbius discrete-event simulator, those values are stored as a simple contiguous allocation of memory with a certain size. Since that storage method is the most space-efficient way to represent the entire dataset, the raw memory is sent directly to the Möbius communication layer to be forwarded to the front-end visualization interface. The front-end, which has a mirrored version of the data structure representing the back-end model object, is responsible for parsing the raw data.

In addition to values of the state variables of the AFI implementation of the model, the current model state also comprises the current future events list. Therefore, the list must also be forwarded to the front-end visualization layer. The back-end layer sends the future event list to the communication layer in the form of pairs that include the associated action index and the deterministic simulation time at which it will fire. The communication layer then forwards that information to the front-end visualization interface to be parsed. The details of model state analysis in the communication layer and the front-end visualization interface are discussed in Sections 3.2.1 and 3.3, respectively.

3.1.2 Model State Modification

Because of the dependencies between elements of the model state, model state modification, as discussed in Section 2.2, is not as trivial as model state analysis. To simplify its implementation, the user is restricted to modifying only one state variable primitive value, or only one firing time of an event in the future event list, at once. That restriction does not mean that the user could not modify multiple model state elements at one paused point in the simulation. It simply means that the entire model state must adjust to a single modification

before the user can specify another change. The restriction simplifies model state modification, as only the enabling relationships of the model must be considered. The reason is that the affecting relationships occur only when an action is directly modified, a feature that is not available through MSDV.

To modify the model state, the MSDV back-end receives a model state modification message from the front-end visualization interface, as described in Section 3.1.2. The message specifies a state variable primitive and its new value. The MSDV back-end updates the specified state variable with the new value, and reevaluates the status of each of the actions with which the state variable shares an enabling relationship. Consequently, if the status of an action switches from enabled to disabled, then the associated event in the future event list is removed from the list. Similarly, if the status of an action switches from disabled to enabled, then the timing distribution of the action is sampled, and it is added to the future event list. If the status of the action does not change, then it does not modify its associated event, or lack thereof, in the future event list.

In addition to directly modifying state variables, the MSDV feature also allows users to directly modify the firing times of the events in the future event list in the simulation time interval $t \geq$ `currentSimulationTime`. Since those event times are independent of the rest of the model state, no further consideration must be paid by the MSDV back-end.

Note that when an action's associated event is added to the future event list, its timing distribution is sampled. Consequently, if the modification of a state variable results in the removal of the action's event from the future event list, then even if the state variable is modified back to its original value, the overall model state is unlikely to return to the same state. Since the timing of events is based on the statistically random distribution of the actions, the event will be added back to the future event list with a different associated time. However, since the firing time for events in the event list may also be changed through MSDV, the old firing time can be restored if desired.

Although model state modification is a helpful feature in analysis of running simulations, it is important to note that any modifications to a running simulation could result in statistical differences to runs without modification. Thus, simulation batches that utilize model state modification should not be considered the final results of a system model analysis. Rather, these simulation batches should be used to help the analyst determine more appropriate parameters and model designs for a complete model that better describes the complex system under consideration.

3.1.3 Breakpoints and Simulation Stepping

As described in Sections 2.3 and 2.4, the simulation-pausing capabilities are provided through breakpoints and simulation stepping. The back-end MSDV contributes to that capability by determining the point at which to stop, and by waiting for further instructions from the front-end visualization interface. The evaluation of both explicit and implicit (stepping) breakpoints occurs in the back-end, rather than the front-end, to eliminate the need to forward the entire model state to the front-end after the firing of each event. Thus, the simulation can proceed at near-optimal solution speed until a breakpoint is hit. Both breakpoint and simulation-stepping messages are discussed in Section 3.2.

3.2 Communication Layer

The communication layer of the Möbius simulator is responsible for providing the medium between the back-end Möbius simulation processes and the front-end visualization interface. This layer operates by forwarding TCP/IP messages between the POSIX sockets of each of those end layers. Each of those messages is represented as a raw byte string, and is parsed by the receiving end layer. The several message types available in the MSDV feature are discussed throughout this section.

3.2.1 Model State Message

The model state message contains a serialized representation of the entire current model state to be forwarded from the back-end Möbius simulation processes to the front-end visualization interface. This message contains the number of state variables in the model, the offset of each state variable in the state variable data, the contents of the state variable data, the number of events in the future event list, the unique index of the action associated with each event in the future event list, and the simulation time t at which the given event will fire, as seen in Table 3.1.

Continuing with the running example presented in Chapter 2, consider the initial model state:

```
SV1->Mark() = 2
SV2->Mark() = 0
SV3->Mark() = 0
```

Table 3.1: Model State Message Protocol

Message	Message Type	Num SVs	Offset 2	...	Offset n	SV Data
Type	`char`	`int`	`int`	...	`int`	`char[]`
Size(bytes)	1	4	4	...	4	`sizeof(SVData)`
Message	Num Actions	E[1].actn	E[1].time	...	E[n].actn	E[n].time
Type	`int`	`int`	`double`	...	`int`	`double`
Size (bytes)	4	4	8	...	4	8

Table 3.2: Model State Message Example

Message	**Byte String**							
3 SVs	0x00	0x00	0x00	0x03				
SV `off[1]`	0x00	0x00	0x00	0x02				
SV `off[2]`	0x00	0x00	0x00	0x02				
SV data	0x00	0x02	0x00	0x00	0x00	0x00		
2 Events	0x00	0x00	0x00	0x02				
`E[0]`	0x00	0x00	0x00	0x00				
`t=1.00`	0x3f	0xf0	0x00	0x00	0x00	0x00	0x00	0x00
`E[1]`	0x00	0x00	0x00	0x02				
`t=1.75`	0x3f	0xfc	0x00	0x00	0x00	0x00	0x00	0x00

```
FEL = { {A1, 1.00}, {A3, 1.75} }
```

This model state is represented in the model state message in Table 3.2.

3.2.2 Modify State Variable Message

The modify state variable message allows the user to modify specified state variable primitive values. This message, which is forwarded from the front-end visualization interface to the back-end simulation processes, contains the unique index of the state variable under consideration, the memory offset of the primitive value in the contiguous memory representing the entire state variable value, the type of the primitive value to be modified, and the new desired value of the state variable primitive value, as seen in Table 3.3.

This message works by first accessing the memory location of the state variable with

Table 3.3: Modify State Variable Message Protocol

Message	Msg Type	SV Index	Value Offset	Value Type	New Value
Type	`char`	`int`	`int`	`char`	`typeof(newValue)`
Size (bytes)	1	4	4	1	`sizeof(newValue)`

```
struct myStruct {
        int myInt1;
        short myShort1;
};
```

Figure 3.2: Modify State Variable Example Struct.

Table 3.4: Modify State Variable Message Example

Message	**Byte String**									
`sv[1][4]=5`	0x03	0x00	0x00	0x00	0x01	0x00	0x00	0x00	0x04	0x01
	0x00	0x05								
`sv[1][0]=7`	0x03	0x00	0x00	0x00	0x01	0x00	0x00	0x00	0x00	0x02
	0x00	0x00	0x00	0x07						

the unique state variable index. Next, using the provided offset, the message accesses the exact memory location of the primitive variable to be modified. For a state variable that is composed of a single primitive variable, this offset will be 0. However, for more complicated state variables, such as a custom struct or array, the offset will be based on the number and type of variables that are stored ahead of this variable in the state variable data structure. For example, if a state variable contained the type `myStruct`, as in Figure 3.2, and the message was to modify `myShort1`, then the offset would be `sizeof(int)`$= 4$.

The message also contains the type of the new value, which is used to perform the necessary conversion of the new value from the network byte order to the host byte order. The conversion is only necessary on little endian machines, since network byte order is equivalent to big endian order, which is the reverse of little endian order.

Another possible implementation of the modify state variable message involves sending the entire value of the complex state variable type, rather than relying on offsets and primitive data types. That method would be much simpler for big endian machines, but would be more difficult to implement for little endian machines because of the way those machines store variables in the back-end simulation memory. In order to guarantee that data are decoded correctly by the back-end, regardless of architecture, the former, primitive data-based implementation is used.

As an example, consider again the user-defined struct `myStruct` presented in Figure 3.2. Assuming that the value of state variable `sv[1]` is of the type `myStruct`, the messages in Table 3.4 instruct the back-end to modify `myShort1` to 5, and `myInt1` to 7.

Also, consider the model state of the running example presented in Chapter 2 at simulation time $t = 1.25$:

```
SV1->Mark() = 1
```

Table 3.5: Modify State Variable Message: Running Example

Message	**Byte String**									
`sv[0][0]=0`	0x03	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x01
	0x00	0x00								
`sv[1][0]=2`	0x03	0x00	0x00	0x00	0x01	0x00	0x00	0x00	0x00	0x01
	0x00	0x02								

Table 3.6: Modify Future Event List Message Protocol

Message	Message Type	Event Index	New Time
Type	`char`	`int`	`double`
Size (bytes)	1	4	8

```
SV2->Mark() = 1
SV3->Mark() = 0

FEL = { {A2, 1.50}, {A3, 1.75}, {A1, 2.00} }
```

To remove a token from the state variable `SV1` and add one to state variable `SV2`, as in Section 2.2, the front-end visualization interface would send the message from Table 3.5 to the back-end simulation processes. The back-end would parse that message, resulting in the following model state:

```
SV1->Mark() = 0
SV2->Mark() = 2
SV3->Mark() = 0

FEL = { {A2, 1.50} }
```

3.2.3 Modify Future Event List Message

The modify future event list message allows the user to modify the firing time of an event in the future event list to a simulation time in the interval $t \geq$ `currentSimulationTime`. This message contains the index of the event in the future event list, and the new desired time at which the event will fire, as seen in Table 3.6.

Consider the model state of the running example presented in Chapter 2 at simulation time $t = 1.25$:

Table 3.7: Modify Future Event List Message Example

Message	Byte String							
E[2]	0x00	0x00	0x00	0x02				
t=1.35	0x3f	0xf5	0x99	0x99	0x99	0x99	0x99	0x9a

```
SV1->Mark() = 1
SV2->Mark() = 1
SV3->Mark() = 0

FEL = { {A2, 1.50}, {A3, 1.75}, {A1, 2.00} }
```

To modify the firing time of action `A1` from $t = 2.00$ to $t = 1.35$ at this simulation time, as was discussed in Section 2.2, the front-end visualization interface would send the message from Table 3.7 to the back-end simulation processes. The back-end would parse the message and update the future event list, resulting in the following model state:

```
SV1->Mark() = 1
SV2->Mark() = 1
SV3->Mark() = 0

FEL = { {A1, 1.35}, {A2, 1.50}, {A3, 1.75} }
```

3.2.4 Breakpoint Message

The breakpoint message allows the user to forward breakpoint information from the front-end user interface to the back-end simulation process during a paused simulation. That information allows the simulation to pause upon reaching a certain specified state, as described in Section 2.3. The organization of that information is shown graphically as a Unified Modeling Language (UML) model in Figure 3.3.

The breakpoint message is composed of a number of breakpoints $[0 \ldots *]$, each of which reduces to a Boolean value. Each of those values is further reduced with the logical `OR` operator to determine the Boolean value of the overall breakpoint list. If that list value evaluates to `true`, then the breakpoint is hit, and the simulation pauses at that simulation time t. Otherwise, if the value evaluates to `false`, the simulation continues to run, reevaluating the breakpoint both immediately following the firing of an action (resulting in model state changes), and before the firing of an action (to evaluate the simulation time between this

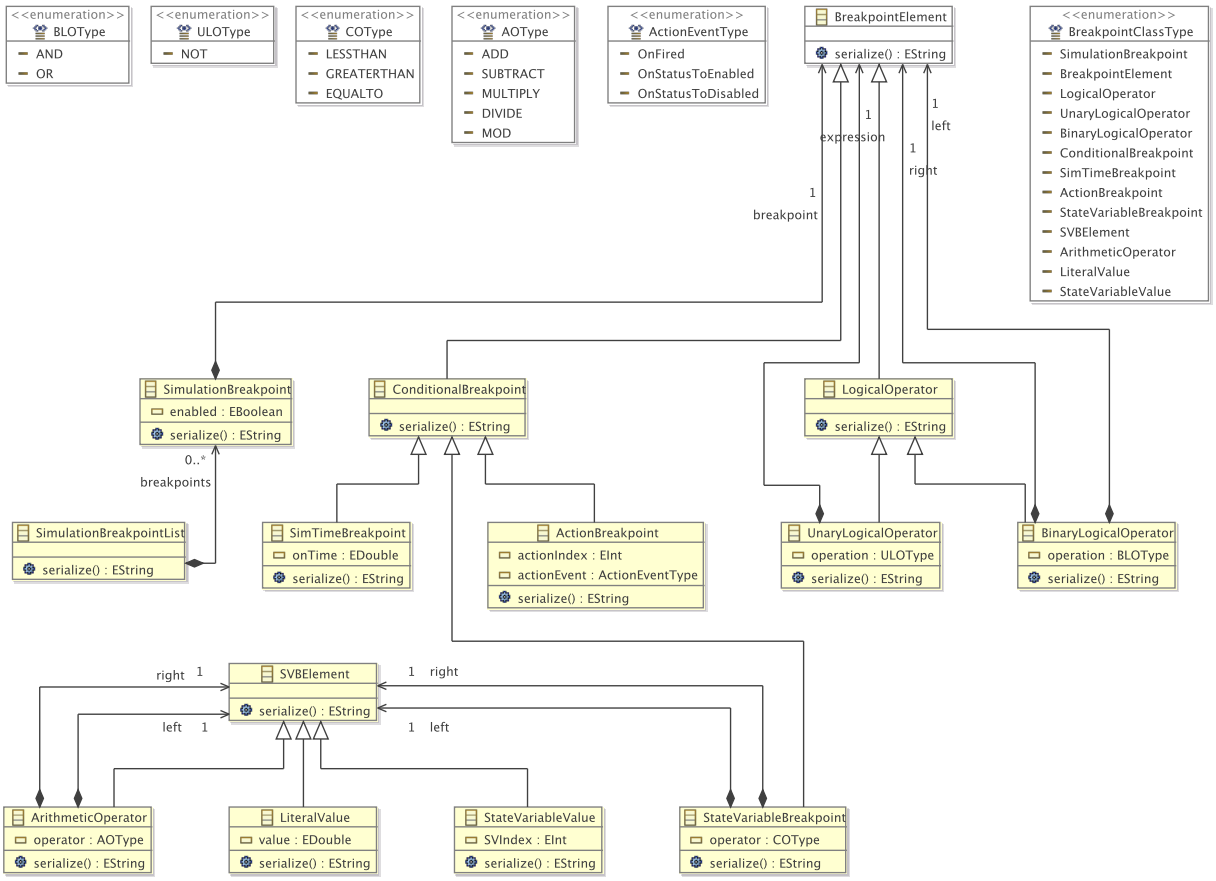


Figure 3.3: Breakpoint Message Protocol UML.

time and the time that the last action fired). The former case potentially results in different action breakpoints and state variable breakpoints, since these breakpoints depend on the model state. The latter case potentially results in varying simulation time breakpoints, since these breakpoints depend on the current simulation time t.

The simulation breakpoint list is the highest level of the breakpoint message protocol. It contains the number of breakpoints, the size of each breakpoint, and the breakpoint information for each breakpoint, as seen in Table 3.8. This information is used to separate the breakpoints in the back-end simulation process, to parse each breakpoint independently.

The next level of the breakpoint message protocol is the breakpoint level, which is either a

Table 3.8: Simulation Breakpoint List

Message	Msg Type	Num BPs	Offset 2	...	Offset n	Simulation Breakpoints
Type	`char`	`int`	`int`	...	`int`	N/A
Size (bytes)	1	4	4	...	4	`sizeof(breakpoints)`

Table 3.9: Simulation Time Breakpoint

Message	Class Type	Break Time
Type	`char`	`double`
Size (bytes)	1	8

Table 3.10: Action Breakpoint

Message	Class Type	Action Event Type	Action Index
Type	`char`	`char`	`int`
Size (bytes)	1	1	4

conditional breakpoint or a logical operator that compares multiple conditional breakpoints. Effectively, this level could be a simulation time breakpoint, an action breakpoint, a state variable breakpoint, a unary logical operator, or a binary logical operator.

A simulation time breakpoint determines whether the specified simulation time is greater than or equal to the current simulation time. More specifically, that breakpoint's message would specify that the simulation pause once the specified simulation time has been reached. For example, if the message specifies a breakpoint at simulation time $t = 5.5$, the breakpoint would return `true` whenever the breakpoint is evaluated at simulation time $t \geq 5.5$. The message simply contains the simulation break time as a double type, as in Table 3.9.

An action breakpoint determines whether a specified action event has just occurred. Such action events include firing of the specified action (`OnFired`), switching of the specified action's status from disabled to enabled (`OnStatusToEnabled`), and switching of the specified action's status from enabled to disabled (`OnStatusToDisabled`). An action breakpoint's message contains the unique action index of the specified action and the action event type, as in Table 3.10.

A state variable breakpoint returns the Boolean value of the comparison operation between the specified left and right state variable operands. The operands, which are cast as `double` values before comparison, are composed of primitive type values in state variable value data structures, literal numerical values, and arithmetic operations between the previous two operands. The highest level of a state variable breakpoint message contains the comparison operator, the left and right operands, and the size of the left operand as an offset for the beginning of the right operand in the serialized message, as in Table 3.11.

A state variable value is represented as the primitive type of the specified value in the state variable data structure, the unique state variable index, and the offset of the primitive specified value of the state variable, as in Table 3.12. In a state variable represented as a single type, the offset is 0.

Table 3.11: State Variable Breakpoint

Message	Class Type	Operation	Right Offset	Left	Right
Type	`char`	`char`	`int`	N/A	N/A
Size (bytes)	1	1	4	`sizeof(left)`	`sizeof(right)`

Table 3.12: State Variable Value

Message	Class Type	SV Type	State Variable Index	Offset
Type	`char`	`char`	`int`	`int`
Size (bytes)	1	1	4	4

A literal value is simply represented as a double, as in Table 3.13.

An arithmetic operator allows the message to specify arithmetic operations between primitive state variable values and literal values. For example, one might want a breakpoint when a certain state variable value `sv1->Mark()` is double the value of another state variable value `sv2->Mark()`. The arithmetic operator allows the message to multiply `sv1->Mark()` by 2 before being evaluated by the comparison operator of the state variable breakpoint level. The operations addition (+), subtraction (−), multiplication (∗), and division (/) are all performed as `double` operations. The mod operation (`%`) is performed as an `int` operation, and the result is cast into a `double` value. The arithmetic operator message fragment contains the specified arithmetic operation, the left and right operands, and the size of the left operand as an offset for the beginning of the right operand in the serialized message, as in Table 3.14.

A unary operator allows a contained breakpoint message to be evaluated as an expression of the specified operator. Although the only unary operator implemented in this protocol is the `NOT` operator, the operator is still explicitly specified to make it easy to add additional unary operators to future implementations. The unary operator message fragment contains the unary operation and the breakpoint expression to be evaluated, as in Table 3.15.

A binary operator allows the message to logically combine the results of two breakpoint expressions, including both the `AND` operation and the `OR` operation. The binary operator message fragment contains the binary operation, the left and right breakpoint expressions, and the size of the left breakpoint as the offset of the beginning of the right breakpoint

Table 3.13: Literal Value

Message	Class Type	Value
Type	`char`	`double`
Size (bytes)	1	8

Table 3.14: Arithmetic Operator

Message	Class Type	Operation	Right Offset	Left	Right
Type	`char`	`char`	`int`	N/A	N/A
Size (bytes)	1	1	4	`sizeof(left)`	`sizeof(right)`

Table 3.15: Unary Logical Operator

Message	Class Type	Operation	Expression
Type	`char`	`char`	N/A
Size (bytes)	1	1	`sizeof(expression)`

expression in the serialized message, as in Table 3.16.

As an example, consider the breakpoint represented in Figure 3.4. In this example, the simulation is to pause when the simulation time reaches $t = 4.25$, the condition that the value of `sv[2][5]` $< 7 +$ `sv[1][4]`[1] after the simulation time reaches $t = 3.1$ is `false`, or `action[1]` fires. The corresponding breakpoint message is listed in Table 3.17.

Consider, again, the running example presented in Chapter 2. Specifying the three breakpoints presented in Sections 2.3.1—2.3.3, we arrive at the breakpoint in Figure 3.5. This breakpoint would result in the message presented in Table 3.18. Applying the breakpoint at the beginning of the simulation would pause the batch at simulation time $t = 1.00$, at which time action `A2` would become enabled, the first of the three conditions to evaluate to `true`.

3.2.5 Step Message

The step message allows a user to continue a simulation until immediately after the next event in the future event list fires. This simple message type, which is forwarded from the front-end visualization interface to the back-end Möbius simulation processes, contains no additional parameters, as seen in Table 3.19. Although this message could be represented explicitly as a breakpoint message combining all action fire events with the `OR` logical operator, this implicit

Table 3.16: Binary Logical Operator

Message	Class Type	Operation	Right Offset	Left	Right
Type	`char`	`char`	`int`	N/A	N/A
Size (bytes)	1	1	4	`sizeof(left)`	`sizeof(right)`

[1] As described in Section 3.2.2, the first index of the state variable corresponds to the unique index of the state variable in the model, and the second index corresponds to the state variable primitive offset within the state variable data structure.

```
(SimTimeBreakpoint 4.25)
(UnaryOperator NOT
  (BinaryOperator AND
    (SimTimeBreakpoint 3.1)
    (StateVariableBreakpoint <
      (StateVariableValue short 2 5)
      (ArithmeticOperator +
        (LiteralValue 7)
        (StateVariableValue double 1 4)
      )
    )
  )
)
(ActionBreakpoint OnFired 1)
```

Figure 3.4: Breakpoint Message Example Pseudo-message.

Table 3.17: Breakpoint Message Example

Message	**Byte String**									
3 BPs	0x00	0x00	0x00	0x03						
`off[1]`	0x00	0x00	0x00	0x09						
`off[2]`	0x00	0x00	0x00	0x3a						
ST 4.25	0x06	0x40	0x11	0x00	0x00	0x00	0x00	0x00	0x00	
`NOT`	0x03	0x00								
`AND`	0x04	0x00	0x00	0x00	0x00	0x09				
ST 3.1	0x06	0x40	0x08	0xcc	0xcc	0xcc	0xcc	0xcc	0xcd	
$<$	0x08	0x00	0x00	0x00	0x00	0x0a				
`sv[2][5]`	0x0c	0x01	0x00	0x00	0x00	0x02	0x00	0x00	0x00	0x05
$+$	0x0a	0x00	0x00	0x00	0x00	0x09				
7	0x0b	0x40	0x1c	0x00	0x00	0x00	0x00	0x00	0x00	
`sv[1][4]`	0x0c	0x03	0x00	0x00	0x00	0x01	0x00	0x00	0x00	0x04
`a[3].fire`	0x07	0x00	0x00	0x00	0x00	0x02				

```
(SimTimeBreakpoint 1.25)
(ActionBreakpoint OnStatusToEnabled 1)
(StateVariableBreakpoint >
  (StateVariableValue short 2 0)
  (StateVariableValue short 0 0)
)
```

Figure 3.5: Breakpoint Message Running Example Pseudo-message.

Table 3.18: Breakpoint Message Running Example

Message	**Byte String**									
3 BPs	0x00	0x00	0x00	0x03						
`off[1]`	0x00	0x00	0x00	0x09						
`off[2]`	0x00	0x00	0x00	0x06						
ST 1.25	0x06	0x3f	0xf4	0x00	0x00	0x00	0x00	0x00	0x00	
`a[1].enabled`	0x07	0x01	0x00	0x00	0x00	0x01				
>	0x08	0x01	0x00	0x00	0x00	0x0a				
`sv[2][0]`	0x0c	0x01	0x00	0x00	0x00	0x02	0x00	0x00	0x00	0x00
`sv[0][0]`	0x0c	0x01	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00

Table 3.19: Step Message

Message	Class Type
Type	`char`
Size(bytes)	1

message type is simpler to use and requires less communication overhead to accomplish this frequently useful operation.

3.3 Front-end Visualization

The front-end visualization and user interface level is where the user interacts with the Möbius framework. That level allows users to define models, run analyses on the models, and view the results of the analyses. As far as the MSDV feature is concerned, that level is where the user can view the running simulation model states at specified simulation time points, and alter the given model states at these simulation time points. In designing that level of the MSDV feature, the goals of modularity and full implementation for specific modeling formalisms, as discussed in Section 1.5, had to be considered. Therefore, the Abstract Functional Interface (AFI) implementation will first be discussed, in Section 3.3.1, as a means of achieving a modular design. Next, implementations of specific modeling formalisms will be discussed, starting with atomic modeling formalisms in Section 3.3.2, and continuing to composed modeling formalisms in Section 3.3.3. A general description of how to implement a new visualization and user interface for additional modeling formalisms is given in Appendix A.

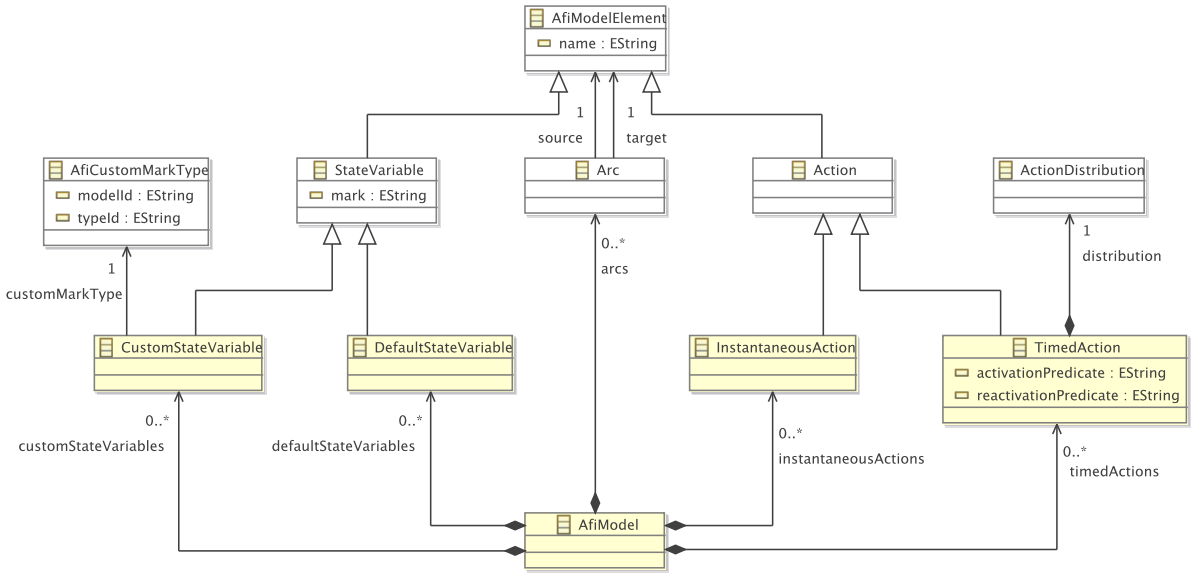


Figure 3.6: AFI Editor UML.

```
context Arc inv:source.oclIsTypeOf(StateVariable) <>
        target.oclIsTypeOf(StateVariable)
context Arc inv:source.oclIsTypeOf(Action) <>
        target.oclIsTypeOf(Action)
context DefaultStateVariable inv:CanParseAsShort(mark)
context CustomStateVariable inv:
        CanParseAs<customMarkType.toType()>(mark)
```

Figure 3.7: AFI Editor OCL.

3.3.1 Abstract Functional Interface

The Abstract Functional Interface (AFI) provides the basis for all other atomic and composed models in the Möbius framework. More specifically, all atomic and composed models are inherited from the parent AFI model class. Although AFI models could theoretically be directly created, current releases of Möbius do not support this functionality, since the AFI is intended simply to be a means of representing higher-level modeling formalisms in a generic and uniform way in the back-end analysis processes. However, in discussing the visualization and user interface of the AFI in the MSDV feature, it is helpful to precisely describe what the AFI editor would be if it were implemented in Möbius. That being said, the AFI editor in the Möbius framework would be defined as shown in the UML diagram and the OCL description in Figures 3.6 and 3.7, respectively.

This hypothetical editor would allow users to define five types of objects: default state variables, custom state variables, timed actions, instantaneous actions, and directed arcs.

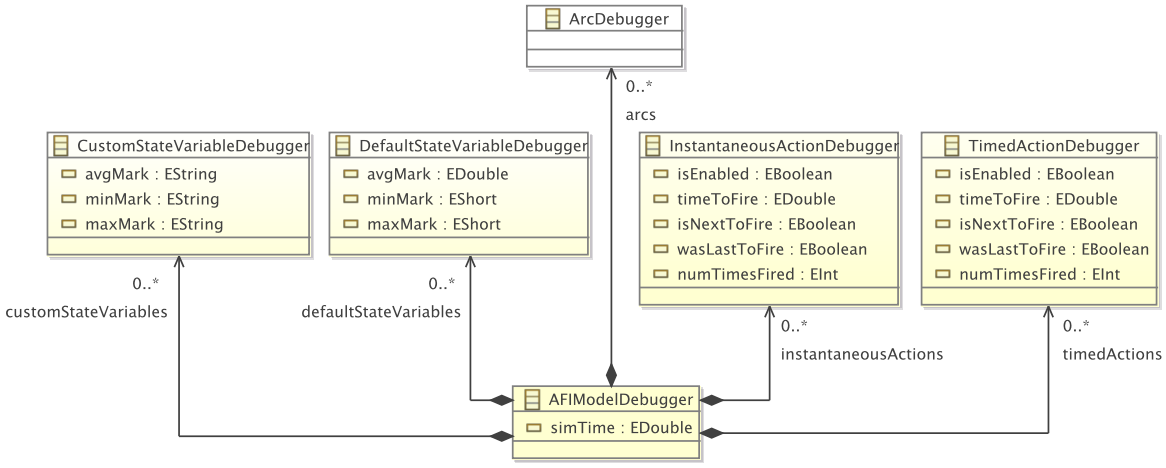


Figure 3.8: AFI Debug and Visualization Editor UML.

The four types of objects that are inherited from the `AFIModelElement` could be placed independently, and the directed arcs would form connections between `StateVariable` type objects and `Action` type objects, in either direction. Although the directed arc objects are not directly represented in an AFI model, they provide the user with an intuitive way to apply enabling and affecting relationships between the state variables and actions in the model. Those relationships are discussed in Section 2.2.

Inherited directly from the classes of the AFI editor, the AFI debug and visualization editor of the MSDV feature expands the functionality and visibility of the editor from the initial static AFI model to the running simulation of the model. That additional functionality is apparent in the inherited UML model displayed in Figure 3.8. In that diagram, each of the five types of objects inherits from the respective class without the `Debugger` suffix in Figure 3.6. Note that the `AFIModelDebugger` class is not a child of the `AFIModel` class. The OCL description of the AFI debug and visualization editor is listed in Figure 3.9.

In the UML and OCL definitions of Figures 3.8 and 3.9, the state variable debugger classes have been expanded to include `minMark`, `maxMark`, and `avgMark` fields. Each of those fields stores the minimum value that the mark becomes, the maximum value that the mark becomes, and the average value of the mark, respectively, over the simulation time interval $t = [0, \texttt{AFIModelDebugger->simTime}]$. Since the `DefaultStateVariable` class simply contains a `short` value, the `minMark` and `maxMark` fields can also be represented as `short` values. The `avgMark` field, on the other hand, must be represented as a real number, since the result of the average calculation is not constrained to a whole number. Thus, the `avgMark` field is represented as a `double` value. Similarly, for the `CustomStateVariableDebugger` class, the `minMark` and `maxMark` fields can be stored as the same type as the mark of the parent

```
context CustomStateVariableDebugger inv:
        CanParseAs<customMarkType.toType()>(minMark)
context CustomStateVariableDebugger inv:
        CanParseAs<customMarkType.toType()>(maxMark)
context CustomStateVariableDebugger inv:
        CanParseAs<customMarkType.toDoubleType()>(avgMark)
context AFIModelDebugger inv: Set{
                instantaneousActions->forAll(isNextToFire),
                timedActions->forAll(isNextToFire)
        }->size() <= 1
context AFIModelDebugger inv: Set{
                instantaneousActions->forAll(wasLastToFire),
                timedActions->forAll(wasLastToFire)
        }->size() <= 1
```

Figure 3.9: AFI Debug and Visualization Editor OCL.

`CustomStateVariable` class. The `avgMark` field, on the other hand, is stored as similar type of the `CustomStateVariable` class in which all of the primitive fields are represented as `double` types.

One note on the scalability of the additional state variable debugger data structures: they more than quadruple the original storage size of the model state. The reasoning is that in the context of both `DefaultStateVariableDebugger` and `CustomStateVariableDebugger`, `sizeof(mark)` = `sizeof(minMark)` = `sizeof(maxMark)` $\leq$ `sizeof(avgMark)`, each of those variables is stored independently of the others. Although that could pose a problem for models with exceptionally large model states, it typically is not an issue with simulation analysis, since each state variable must be explicitly defined during model creation. That differs in the approaches of other analysis techniques, such as transient solvers, but those other techniques fall outside the scope of the MSDV feature.

Also, in the UML and OCL definitions of the AFI debug and visualization editor, the action debugger classes have been expanded to include `isEnabled`, `timeToFire`, `isNextToFire`, `wasLastToFire`, and `numTimesFired` fields. The `isEnabled` field contains a `Boolean` value that indicates whether the action is enabled or disabled. The `timeToFire` field is a `double` value that stores the simulation time t at which the enabled action will fire next, as retrieved from the future event list data. If the action is disabled, and hence not in the future event list, the `timeToFire` takes the invalid simulation time value of $t = -1.0$. The `isNextToFire` and `wasLastToFire` fields contain `Boolean` values that indicate whether the action is referenced by the first event in the future event list and whether the action is the last one to have fired, respectively. As the OCL definition in Figure 3.9 shows, the total number of actions in the `AFIModelDebugger` with those values marked as true can only be 0 or 1. In most cases, only one activity at a time will have either of these values marked as true, as there is only one

first item in the future event list, and only one action that has just fired, as multiple actions are not represented as simultaneous in the Möbius simulation framework. However, if the future event list is empty, such as at the end of a simulation, then none of the actions will be marked as `isNextToFire`. Also, at the beginning of the simulation, before any actions have fired, the `wasLastToFire` field will be `false` for all actions. The `numTimesFired` field simply stores, as an `int` value, the number of times the action has fired during the simulation time interval $t = [0, \texttt{AFIModelDebugger->simTime}]$.

3.3.2 Atomic Models

As discussed in Section 1.5.2, one of the goals of the project presented in this thesis is to implement a visualization interface for several specific atomic modeling formalisms. Each of the modeling-formalism-specific visualization interfaces inherits the functionality of the AFI visualization interface, discussed in Section 3.3.1, in addition to the functionality specific to the given formalism. The implementations of stochastic activity networks (SAN) and ADversary VIew Security Evaluation (ADVISE) models are discussed next.

Stochastic Activity Networks

The SAN visualization interface extends the functionality of the AFI visualization interface to represent data in the form of SAN elements, of which the original model under consideration is composed. Specifically, the SAN visualization interface is composed of `Place`, `ExtendedPlace`, `TimedActivity`, `InstantaneousActivity`, `Arc`, `InputGate`, and `OutputGate` elements, the first five of which are directly inherited from the AFI `DefaultStateVariable`, `CustomStateVariable`, `TimedAction`, `InstantaneousAction`, and `Arc` elements, respectively. Although the last two `SAN` elements, `InputGate` and `OutputGate`, affect the model during simulation, they themselves are static elements that do not change during the simulation. Therefore, they can be displayed to the user as static elements, allowing the SAN visualization interface to rely on the AFI parent methods to perform the majority of the necessary functionality.

It is helpful first to define the SAN model editor. The UML and OCL definitions of this editor are shown in Figures 3.10 and 3.11, respectively. The inheritance relationships between the SAN Editor and the AFI Editor are shown in Table 3.20.

In the UML and OCL definition of the SAN editor, the user is restricted to adding only the following elements to a model: `Place`, `ExtendedPlace`, `InstantaneousActivity`, `TimedActivity`, `InputGate`, `OutputGate`, `SanArc`, and `ActivityCase`. The first six of those

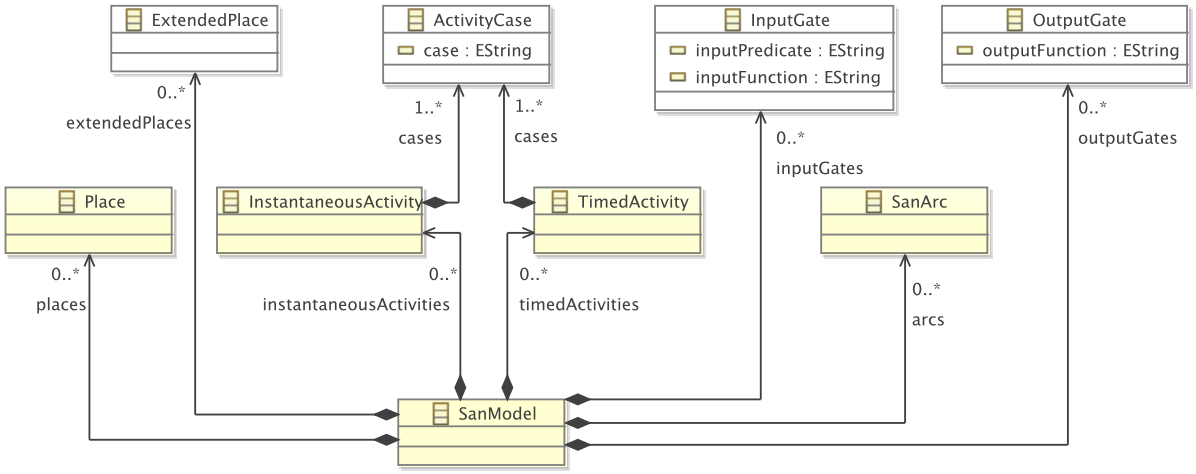


Figure 3.10: SAN Editor UML.

```
context SanArc inv: if (source.type == Place)
        {(target.type == InputGate) || (target.type == Action)}
context SanArc inv: if (source.type == ExtendedPlace)
        {(target.type == InputGate)}
context SanArc inv: source.type <> Action
context SanArc inv: if (source.type == ActivityCase)
        {(target.type == OutputGate) || (target.type == Place)}
context SanArc inv: if (source.type == InputGate)
        {(target.type == Action)}
context SanArc inv: if (source.type == OutputGate)
        {(target.type == StateVariable)}
```

Figure 3.11: SAN Editor OCL.

Table 3.20: SAN to AFI UML Inheritance

Child Class	**Parent Class**
Place	DefaultStateVariable
ExtendedPlace	CustomStateVariable
InstantaneousActivity	InstantaneousAction
TimedActivity	TimedAction
ActivityCase	AfiModelElement
SanArc	Arc
InputGate	AfiModelElement
OutputGate	AfiModelElement

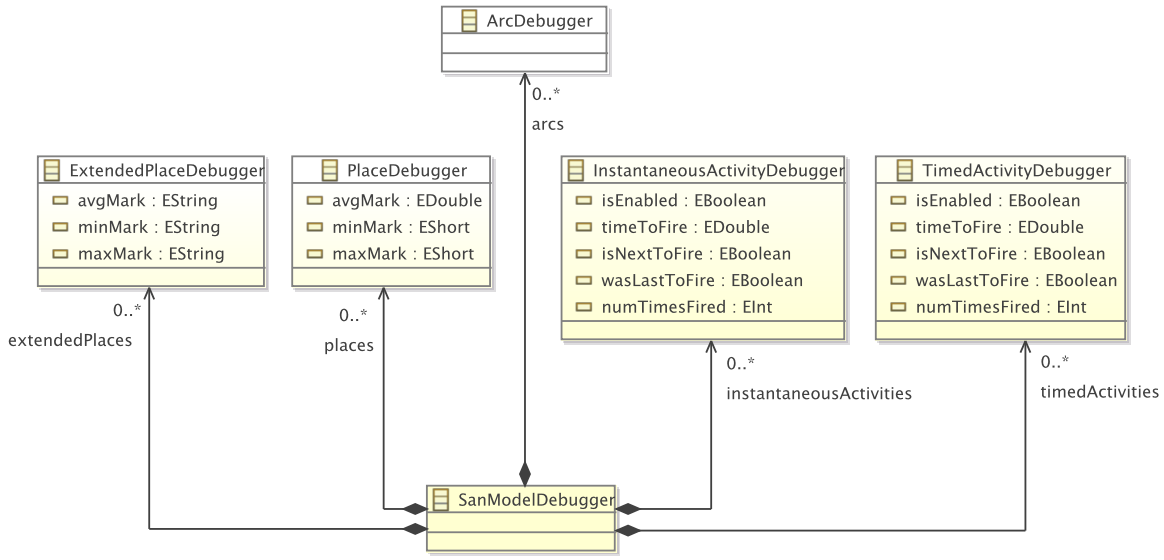


Figure 3.12: SAN Debug and Visualization Editor UML.

elements can be directly added to the SAN model. The `SanArc` element, like the `Arc` element in the AFI editor, is a directed arc that can only be used to connect model elements, and is limited to the restrictions imposed by the OCL definition in Figure 3.11. `ActivityCase` elements can only be directly added to either an `InstantaneousActivity` or a `TimedActivity`.

Just as the classes of the AFI debug and visualization editor inherit directly from those of the AFI editor, the modeling element classes of the SAN debug and visualization editor also inherit directly from those of the SAN editor, excluding the SanModelDebugger class. The UML and OCL definitions of the SAN debug and visualization editor are shown in Figures 3.12 and 3.13, respectively.

The additional fields of the SAN debug and visualization editor show that its functionality is similar to that of the AFI debug and visualization editor. Specifically, each `Place` element, both normal and extended, also contains the additional `minMark`, `maxMark`, and `avgMark` fields, and each `Activity` element, whether timed or instantaneous, also contains the `isEnabled`, `timeToFire`, `isNextToFire`, `wasLastToFire`, and `numTimesFired` fields. Each of those fields behaves like the corresponding field in the AFI version of the editor.

One key difference between the SAN debug and visualization editor and the AFI debug and visualization editor is that in addition to representing the model state data as text, the SAN visualization interface also includes the option to display each SAN model element as a visual representation of its current contents. For example, the intensity of the color of a `Place` visual element is associated with the number of tokens currently contained by the associated `Place` element, increasing as it gains more tokens. Also, each `Activity` visual

```
context ExtendedPlaceDebugger inv:
        CanParseAs<customMarkType.toType()>(minMark)
context ExtendedPlaceDebugger inv:
        CanParseAs<customMarkType.toType()>(maxMark)
context ExtendedPlaceDebugger inv:
        CanParseAs<customMarkType.toDoubleType()>(avgMark)
context SanModelDebugger inv: Set{
                instantaneousActivities->forAll(isNextToFire),
                timedActivities->forAll(isNextToFire)
        }->size() <= 1
context SanModelDebugger inv: Set{
                instantaneousActivities->forAll(wasLastToFire),
                timedActivities->forAll(wasLastToFire)
        }->size() <= 1
```

Figure 3.13: SAN Debug and Visualization Editor OCL.

element is highlighted with a different color to indicate if it was the last `Activity` to fire (red) or will be the next `Activity` to fire (green). That additional visualization functionality helps simulation analysts quickly understand the current state of the model under investigation. Those features are examined in action in the case study presented in Section 4.2.

ADVISE

As with the SAN visualization interface, the ADVISE visualization interface also extends the functionality of the AFI visualization interface. The ADVISE visualization interface represents the current model state of the simulation batch through the ADVISE model elements: `Goal`, `Knowledge`, `Skill`, `Access`, `AttackStep`, `Adversary`, `AdversaryArc`, and `AEGArc`. In this representation, although the model state is not directly affected by the `AEGNode` objects, they are displayed to provide a more comprehensive view of the model state to the user.

As with the other visualization interface discussions, we first discuss the ADVISE model editor. The UML and OCL definitions of this editor are shown in Figures 3.14 and 3.15, respectively. The inheritance relationships between the ADVISE Editor and the AFI Editor are shown in Table 3.21.

In this definition, the user can add `Goal`, `Knowledge`, `Skill`, `Access`, `AttackStep`, and `Adversary` elements directly to the model, while `AdversaryArc` and `AEGArc` elements are added as links between the appropriate node elements. More specifically, `AdversaryArc` elements are directed arcs added either between an `AEGNode` element and an `AttackStep` element, or between an `AttackStepOutcome` element and an `AEGNode` element. `AdversaryArc` elements are added between an `Adversary` element and an `AEGNode` element or an

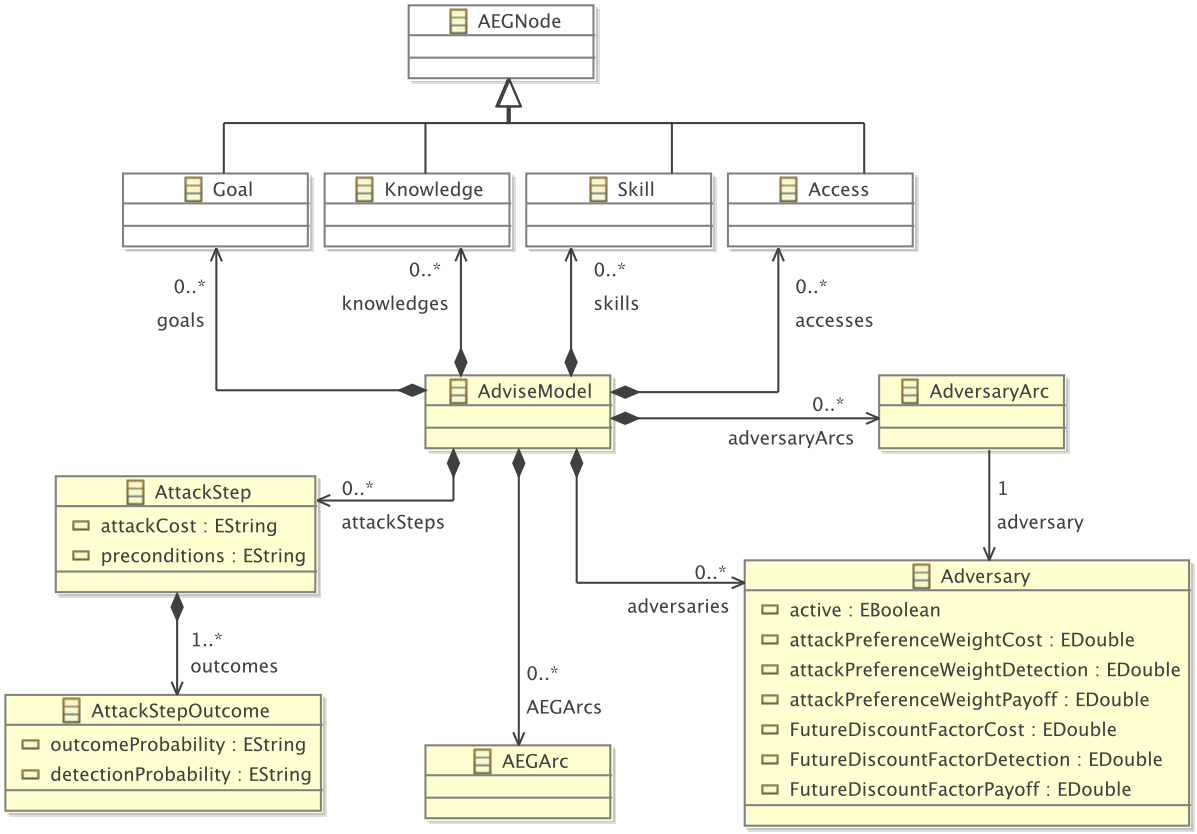


Figure 3.14: ADVISE Editor UML.

```
context AEGArc inv: source.type <> AttackStep
context AEGArc inv: target.type <> AttackStepOutcome
```

Figure 3.15: ADVISE Editor OCL.

Table 3.21: ADVISE to AFI UML Inheritance

Child Class	**Parent Class**
AEGNode	AfiModelElement
AttackStep	TimedAction
AttackStepOutcome	AfiModelElement
AEGArc	Arc
AdversaryArc	DefaultStateVariable
Adversary	AfiModelElement

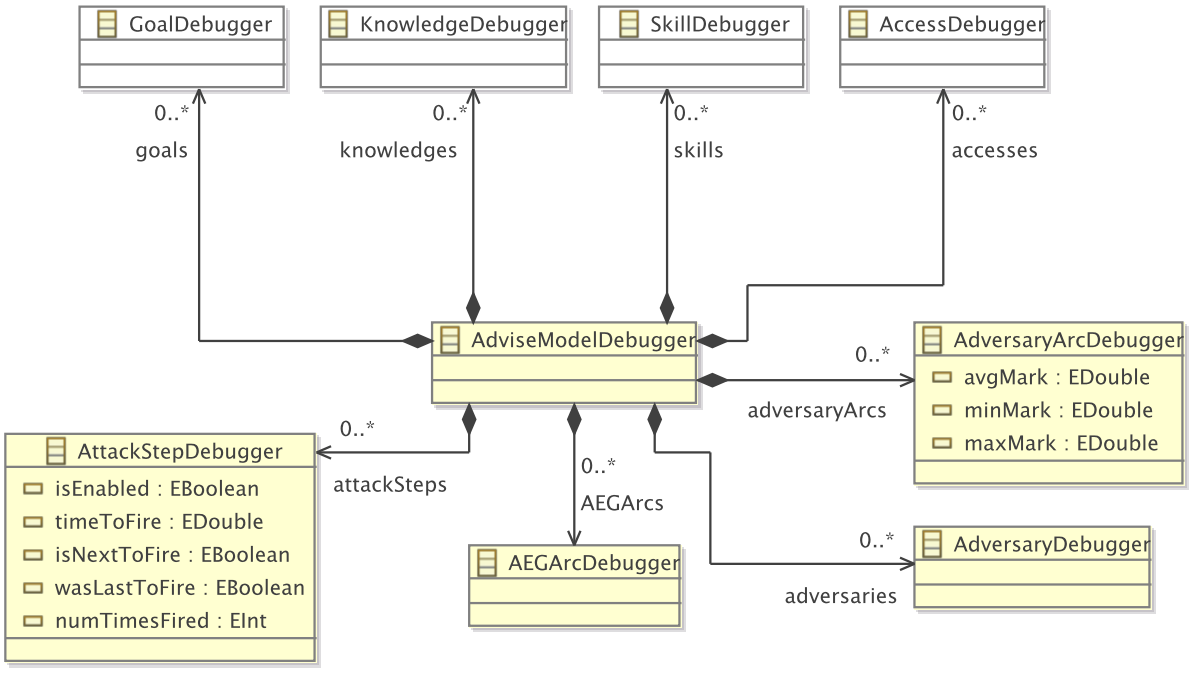


Figure 3.16: ADVISE Debug and Visualization Editor UML.

`AttackStep` element. The `AdversaryArc` element, as seen from Table 3.21, inherits the `DefaultStateVariable` functionality from the AFI editor, and, hence, collectively stores the current model state of the ADVISE model. Similarly, the `AttackStep` element inherits the `TimedAction` functionality from the AFI editor, and, thus, represents the model state transitions throughout each simulation batch.

As with the relationship between the previously mentioned editors and their respective debug and visualization editors, the modeling element classes of the ADVISE debug and visualization editor also inherit directly from those of the ADVISE editor, excluding the AdviseModelDebugger class. The UML definition of the ADVISE debug and visualization editor is shown in Figure 3.16.

Noting from the additional fields of the model elements of the ADVISE debug and visualization editor, the ADVISE debug and visualization editor shares similar functionality with both the AFI and SAN debug and visualization editor. Specifically, each `AdversaryArc` element also contains the additional `minMark`, `maxMark`, and `avgMark` fields, and each `AttackStep` element also contains the `isEnabled`, `timeToFire`, `isNextToFire`, `wasLastToFire`, and `numTimesFired` fields. Each of these fields behaves similarly to those of the AFI and SAN versions of the editor.

One key difference between the ADVISE debug and visualization editor and the AFI debug

and visualization editor is that, in addition to representing the model state, the ADVISE visualization interface also shows attack paths of the different adversaries when they are highlighted by the user.

3.3.3 Composed Models

As discussed in Section 1.5.3, and echoing the atomic modeling formalism goals discussed in the previous section, one of the goals of this project is to implement a visualization interface for a specific composed modeling formalism. The composed modeling formalism on which we focus on in this section is the commonly used Rep/Join modeling formalism.

Rep/Join Model

The Rep/Join state-sharing composed modeling formalism, formerly known as the composed SAN-based reward model (SBRM) [31][32][33], was initially created as a convenient, and more efficient, way to combine SAN submodels into larger system models. However, the submodel specification shifted from SAN submodels to more generic AFI submodels after the AFI was implemented in the Möbius modeling framework. That shift in submodel specification broadened the restriction that only SAN models could be combined, and made it possible to combine any combination of modeling formalisms that inherit from the AFI. With that added flexibility, Rep/Join composed models now provide an effective way to combine any combination of modeling formalisms supported by the Möbius modeling framework, including both atomic and composed modeling formalisms. A UML definition for the Rep/Join composed modeling formalism editor is shown in Figure 3.17.

Note that in the definition given in Figure 3.17, the modeling element classes of the Rep/Join editor, unlike those of atomic modeling editors, are not inherited from the AFI editor element classes. Instead, each `Submodel` element in the Rep/Join editor references another model, either atomic or composed. If each `Submodel` element of a composed model is considered a child node of the parent composed model, the leaves of the tree are guaranteed to be atomic models. Since the modeling element classes of all atomic models are inherited from the AFI modeling element classes, and since the composed model is essentially a composition of these atomic models, the composed model, therefore, is indirectly inherited from the AFI model. Consequently, any composed model can also be represented as an AFI model.

The Rep/Join composed modeling formalism is composed of three modeling elements: `Submodel`, `Rep`, and `Join`. The `Submodel` element represents another user-defined model element of the system represented by the overarching composed model. For example, if the

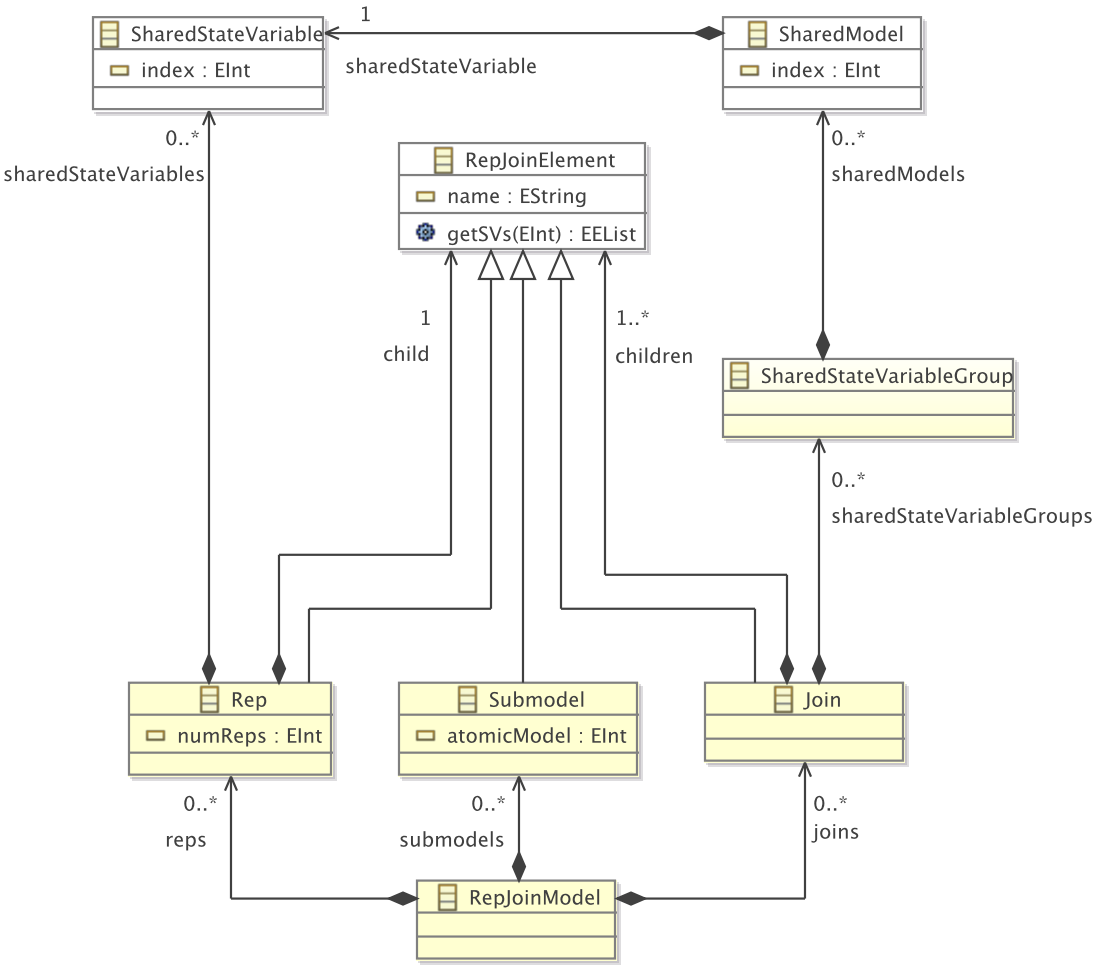


Figure 3.17: Rep/Join Editor UML.

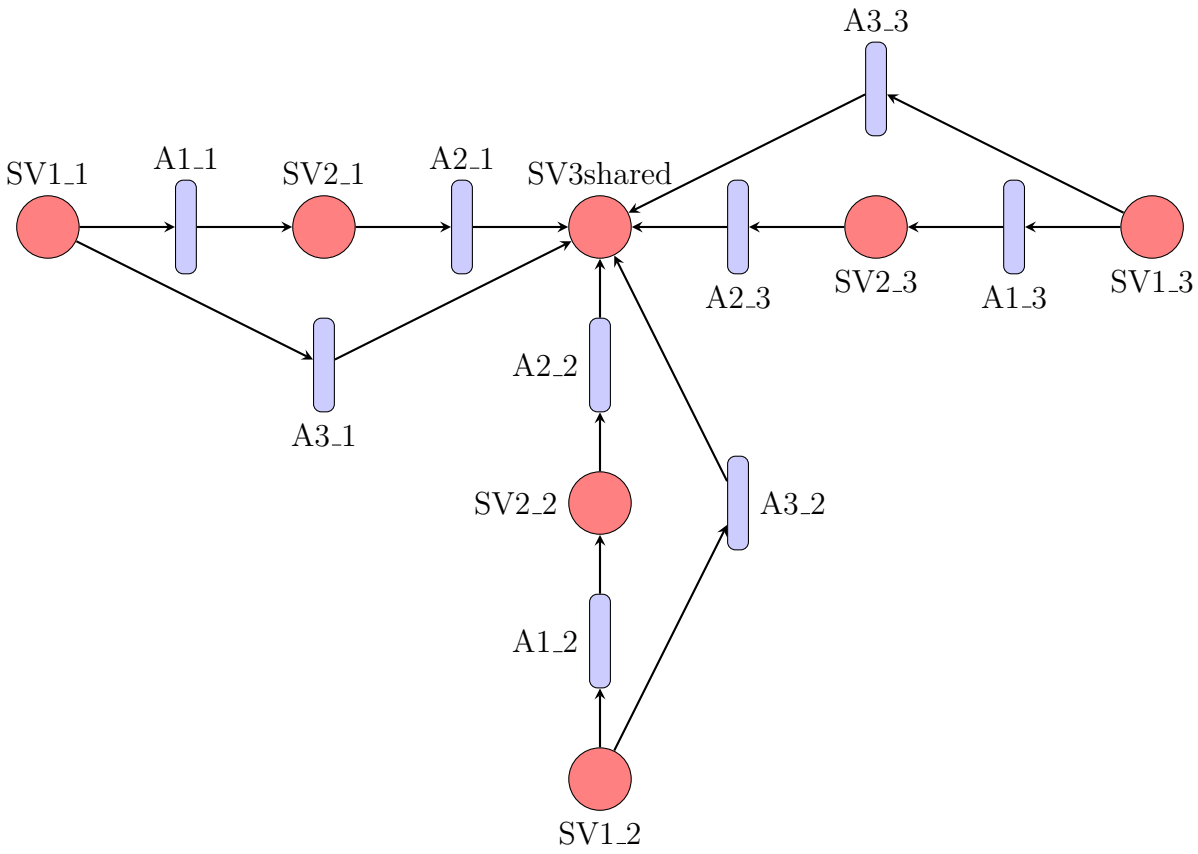


Figure 3.18: AFI Representation of Rep Example.

user is modeling the reliability of a simple computer architecture, the `Submodel` elements may include the atomic models representing a hard drive, a CPU, a system bus, and a NIC. In that case, the composed model represents the entire computer system, composed of those atomic `Submodel` elements. `Submodel` elements can be specified in any modeling formalism supported by the Möbius modeling framework, whether it is an atomic modeling formalism or a composed modeling formalism.

The `Rep` element allows the model to specify a specific number of instances of a child `RepJoinElement`, and the state variables of the child `RepJoinElement` to share among all of the instances. For example, consider a Rep element with three instances of a child of the SAN model presented in Chapter 2, all sharing the state variable `SV3`. In that newly created model, state variable `SV3` across all three instances of the atomic model effectively becomes a single state variable `SV3shared`. An equivalent model created using only the AFI would look like the model defined in Figure 3.18. From the figure, it should be apparent why it quickly becomes cumbersome to specify numerous repetitions of submodels through atomic modeling formalisms alone.

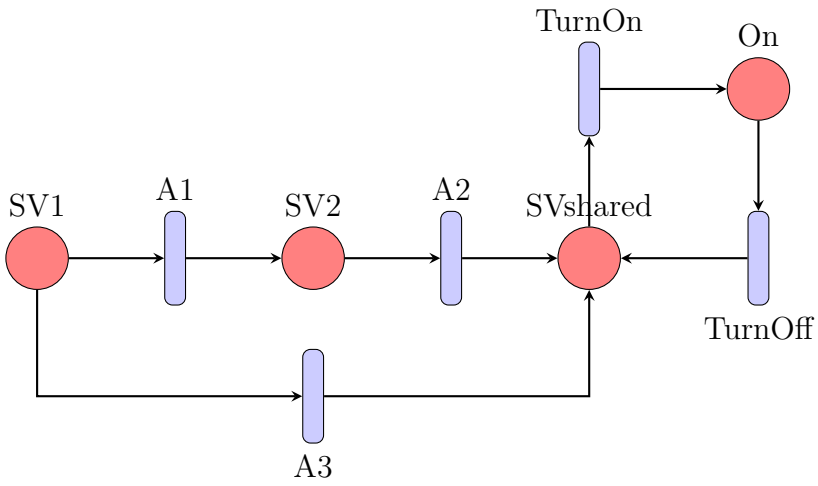


Figure 3.19: AFI Representation of Join Example.

The `Join` element allows the user to specify a group of `RepJoinElements` and the state variables to share among them. For example, consider a `Join` element between the SAN model presented in Section 1.3 and the SAN model presented in Chapter 2, sharing the state variable `Off` from the former model and the state variable `SV3` from the latter. In the newly created model, those two state variables effectively become the single state variable `SVshared`. The token originating in the state variable `SV1` of the latter model gets added to the loop between the `Off` and `On` state variables in the former model. Creating an equivalent model using only the AFI would look like the model shown in Figure 3.19.

Since composed models can be represented as AFI models, the default AFI debug and visualization editor would still be compatible with them. However, since the state variables and actions do not need to be explicitly defined by the user, the size of the models can grow quickly with the `Rep` element. That rapid growth of the AFI version of the model quickly translates into a cumbersome and less useful visualization of the model state. Therefore, the Rep/Join debug and visualization editor maintains the original structure of the Rep/Join model, composed of the `Submodel`, `Rep`, and `Join` modeling elements. UML and OCL definitions for the Rep/Join debug and visualization editor are shown in Figures 3.20 and 3.21, respectively. The inheritance relationships between the Rep/Join debug and visualization editor and the Rep/Join model editor are shown in Table 3.22.

In the Rep/Join debug and visualization editor, the `Submodel` visual element gains the following fields: `wasLastToFire`, `wasLastToFireIndex`, `wasLastToFireActionName`, `isNextToFire`, `isNextToFireIndex`, and `isNextToFireActionName`. The `wasLastToFire` field, which is of the `Boolean` type, represents whether or not the last action that fired occurred within this submodel, or within a set of submodels if this submodel is a child

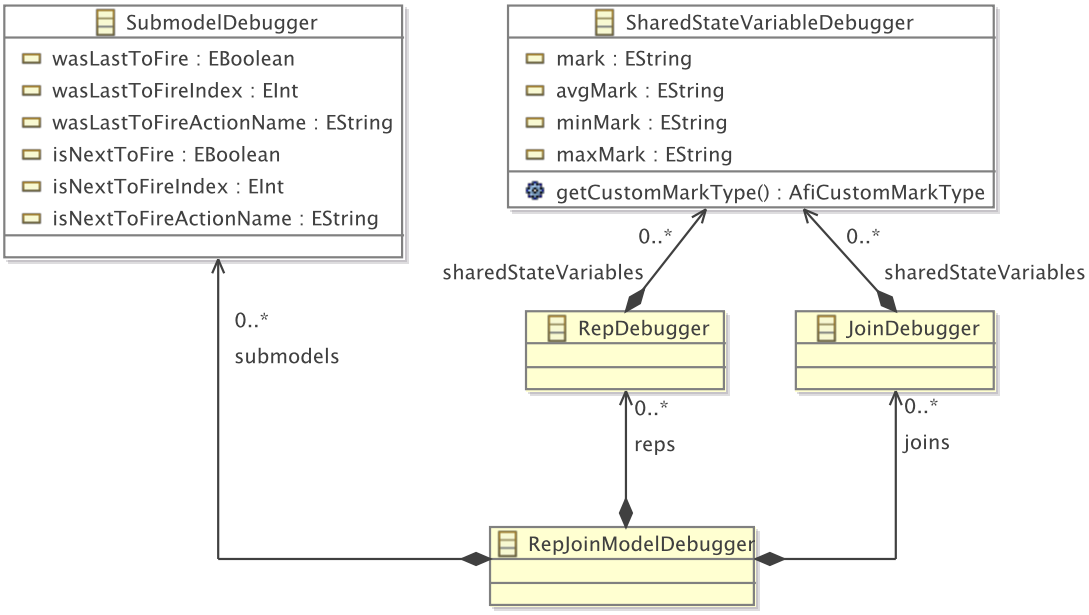


Figure 3.20: Rep/Join Debug and Visualization Editor UML.

```
context SharedStateVariableDebugger inv:
        CanParseAs<getCustomMarkType().toType()>(mark)
context SharedStateVariableDebugger inv:
        CanParseAs<getCustomMarkType().toType()>(minMark)
context SharedStateVariableDebugger inv:
        CanParseAs<getCustomMarkType().toType()>(maxMark)
context SharedStateVariableDebugger inv:
        CanParseAs<getCustomMarkType().toDoubleType()>(avgMark)
context RepJoinModelDebugger inv: Set{
                submodels->forAll(isNextToFire)
        }->size() <= 1
context SanModelDebugger inv: Set{
                submodels->forAll(wasLastToFire)
        }->size() <= 1
```

Figure 3.21: Rep/Join Debug and Visualization Editor OCL.

Table 3.22: Rep/Join Debug and Visualization Editor to Rep/Join Model Editor UML Inheritance

Child Class	Parent Class
SubmodelDebugger	Submodel
RepDebugger	Rep
JoinDebugger	Join

of a `Rep` element. The `wasLastToFireIndex` field, which is of the type `int`, represents the index of the specific submodel in which the last fired action occurred, assuming that `wasLastToFire=true` and the submodel is a child of a `Rep` element. If `wasLastToFire=false`, then the `wasLastToFire` field takes the invalid value of -1. If this submodel is not a child of a `Rep` element, then the `wasLastToFire` field contains either a 0 or a -1, depending on the `wasLastToFire` field. The `wasLastToFireActionName` field, which is of the `String` type, stores the name of the last fired action as long as `wasLastToFire=true`. If `wasLastToFire=false`, then the `wasLastToFire` field assumes the value of an empty string. The `isNextToFire` group of fields is similar to the `wasLastToFire` group of fields, but it pertains to the next action to fire, rather than the last action that has fired.

Also, both the `Rep` and `Join` visual elements gain the `mark` field, which is a mutable field that represents the values of the shared state variables of its child modeling elements. For example, if a given `Rep` element `R1` shares a state variable `SV1` within its child instances, this state variable `SV1` would be an available mutable property of `R1`. The `mark` field works the same way for `Join` elements.

Note that in this implementation of the Rep/Join debug and visualization editor, the added scalability and comprehensibility are traded off against simulation batch mutability. By that, we mean that the user gains a more comprehensive view of the running simulation batch at the expense of needing to edit all of the model state. Specifically, using the Rep/Join and visualization specific editor, the user is able to modify shared state variables only within the `Rep` and `Join` elements. Unshared state variables within specific `Submodel` elements, as well as the future event list in its entirety, are unmutable from this representation. In order to gain greater editorial power, the user could choose to use the AFI debug and visualization editor at the expense of scalability and comprehensibility of large models. That limitation of the current implementation of the editor will be addressed in future work, as discussed in Chapter 5.

CHAPTER 4

CASE STUDY: ATTACK ON AMI

To examine the utility of the new features added to the Möbius discrete-event simulator by the MSDV feature, we will examine a case study that pertains to the real-world scenario of deploying an intrusion detection system (IDS) in an advanced metering infrastructure (AMI) network. An AMI is an infrastructure that enables electricity companies to communicate with their respective electricity meters at remote customer locations. The messages sent through the infrastructure include data such as customer electricity usage information that has been detected by the meter, updated pricing information from the electric company, and alerts about outages in the infrastructure. The goal of an AMI, then, is to simplify and automate the control and visibility of the entire power infrastructure that is under consideration [34].

Clearly, as seen from its introduction alone, an AMI is a powerful infrastructure that increases the ease of operating a large power network. Simultaneously, an AMI also lowers the long-term cost of its maintenance through its increased visibility of the entire infrastructure from a remote location. Despite the advantages of using AMI, however, the additional functionality (in contrast to the functionality of a traditional power infrastructure) opens up the possibility for new failure scenarios of the system. Malicious entities could take advantage of the newly available failure scenarios of the infrastructure and compromise the availability and confidentiality of the critical power systems under consideration. To reap the benefits of the powerful functionality of an AMI while minimizing the failure scenario surface available to malicious attackers, we consider the deployment of an intrusion detection system (IDS) in the AMI network.

The first consideration of the deployment of an IDS in an AMI regards the specific IDS architecture to be employed. The architecture deployment options include the centralized infrastructure, the embedded infrastructure, the dedicated infrastructure, and the hybrid infrastructure. The centralized infrastructure consists of a single IDS sensor deployed at the electric company. The embedded infrastructure consists of embedding an IDS sensor in each deployed smart meter. A dedicated infrastructure consists of the deployment of multiple dedicated IDS sensors geographically within the AMI network. The hybrid infrastructure is

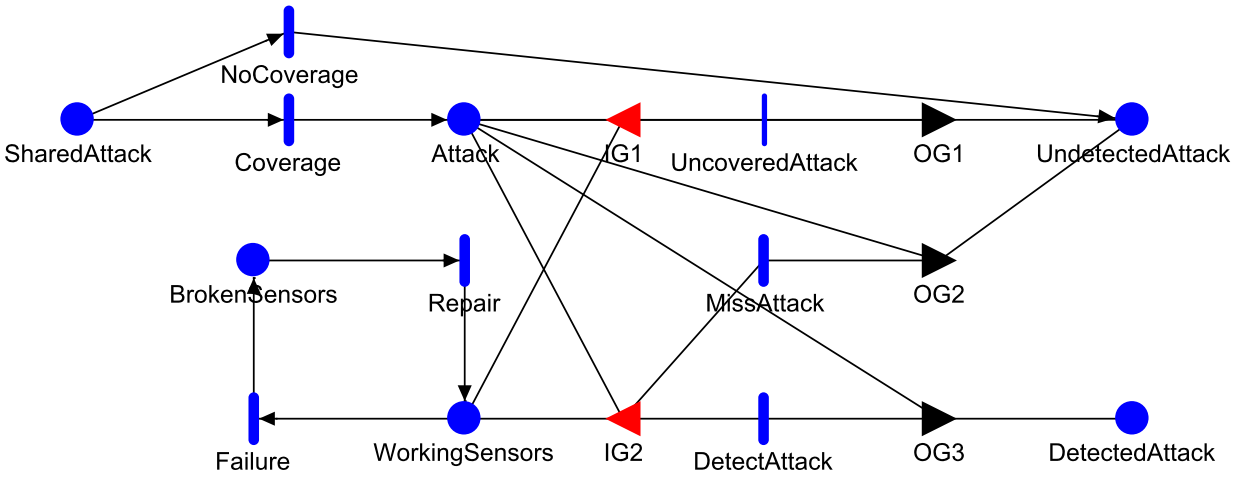


Figure 4.1: SAN: Single Attack on AMI with Dedicated IDS Architecture.

any combination of the three previously mentioned infrastructures. Weighing cost, benefits, and previous considerations from [35], we chose to examine the deployment of a dedicated IDS architecture.

Although dedicated IDS sensors can provide a powerful layer of protection for an AMI network, that type of sensor still inherently possesses some limiting restrictions. One of the restrictions of a dedicated IDS sensor is that the sensor must be physically located within the range of an attack in order for that sensor to detect that attack. If the attack occurs outside of the range of the sensor, then the sensor has a zero percent chance of detecting that attack. Another restriction, inherent to IDS in general, is that the attack must somehow be detectable by the IDS. If the attack is novel enough, there is no way for the IDS to distinguish the attack from typical network traffic. A third, but obvious, limitation is that the sensor can only detect an attack while it is operational. If it is currently unavailable (nonoperational), then it is unable to detect any attack.

Although much consideration has gone into designing an IDS for an AMI network [36][37] [38], we simplify the representation of the IDS to the three restrictions discussed in the previous paragraph. The resulting SAN representation of an attack on the system is pictured in Figure 4.1. In that representation, each IDS sensor in the system is represented as a token residing in either the `WorkingSensors` or `BrokenSensors` places. The position of the token indicates whether the given sensor is currently operational or unavailable. The tokens alternate between the two states through the `Failure` and `Repair` activities, each of which has an exponential rate. The ratio between the rates of the two activities represents the availability of each of the sensors.

The `SharedAttack` place is initialized with a single token that represents an attack on the

AMI network. That token causes a race condition between the `NoCoverage` and `Coverage` activities. The outcome of the race condition represents whether the attack occurred outside or inside the area covered by the IDS network. If the `NoCoverage` activity fires before the `Coverage` activity, then the attack token moves directly to the `UndetectedAttack` place, indicating that the attack was not detected by the IDS network. If the `Coverage` activity fires before the `NoCoverage` activity, then the attack token moves to the `Attack` place to be evaluated by the IDS network. If the attack type is not recognizable to the IDS network, then the attack token moves directly through the `UncoveredAttack` instantaneous activity to the `UndetectedAttack` place. Otherwise, a race condition occurs between the `MissAttack` and `DetectAttack` activities, each of which have an exponential distribution with a rate that relies on the number of tokens currently in the `WorkingSensors` place. If the `MissAttack` activity occurs first (signifying that the IDS sensors that cover the attack location are currently unavailable), the attack token moves to the `UndetectedAttack` place. If the `DetectAttack` activity occurs first, then the attack token moves to the `DetectedAttack` place, indicating that the attack was detected by at least one of the operational IDS sensors.

Throughout this chapter, we examine the utility of the MSDV feature in terms of the previously presented model. We begin by analyzing the model with the default AFI debug and visualization editor in Section 4.1. Next, we analyze the model using the SAN-specific debug and visualization editor in Section 4.2. Finally, we use the original model to create a more complex Rep/Join composed model of a more specific system, and we analyze the resulting model with the Rep/Join debug and visualization editor in Section 4.3.

4.1 SAN with AFI Debugger

Although the model under consideration is specified in the SAN modeling formalism (so the MSDV feature would default to the implemented SAN-specific debug and visualization editor), we first examine the model with the AFI debug and visualization editor for demonstration purposes. The AFI debug and visualization editor, used for any Möbius formalism that does not yet have a specific MSDV implementation, still provides a powerful interface with which to effectively analyze simulation batches for most models. The initial representation of the AFI version of the model is pictured in Figures 4.2 and 4.3. Note that this representation is composed only of AFI modeling elements: state variables, actions, and directed arcs.

First, let us analyze the initial model state of this representation. Focusing on any modeling element (by clicking on it) opens that modeling element's current properties in the

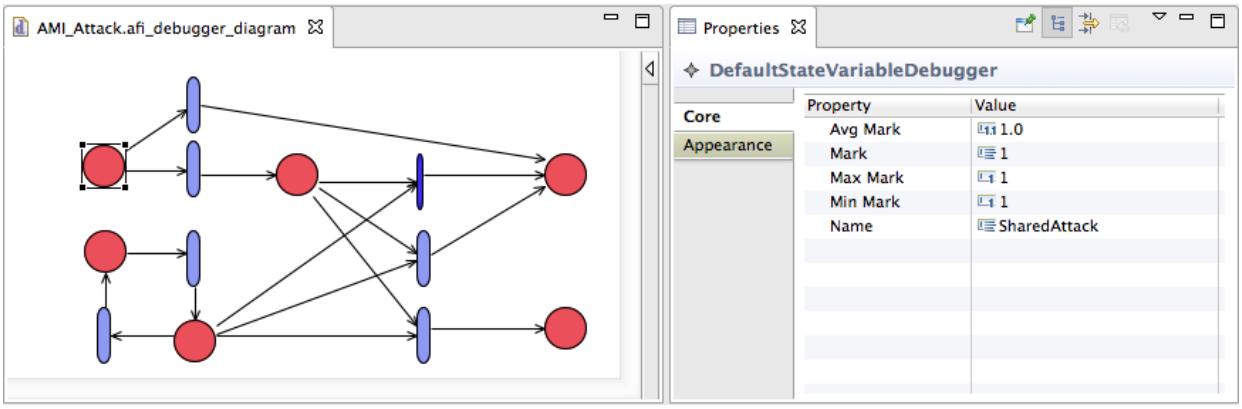


Figure 4.2: SAN Model in AFI Debugging and Visualization Editor.

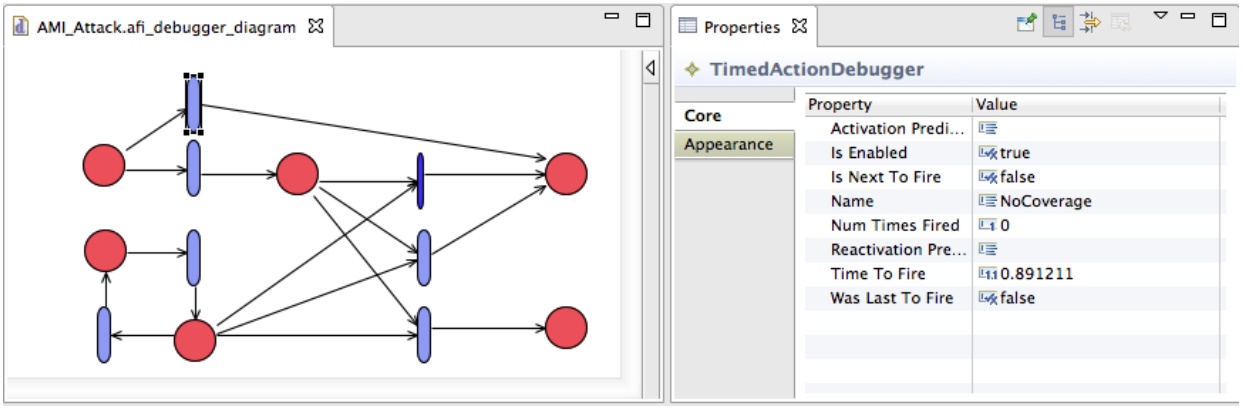


Figure 4.3: SAN Model in AFI Debugging and Visualization Editor.

"Properties" view. The properties are the same as those in the debug and visualization definitions presented throughout section 3.3. Focusing on the `SharedAttack` state variable, the editor shows that `mark` = 1, `minMark` = 1, `maxMark` = 1, and `avgMark` = 1.0. Since no changes have occurred in the simulation yet, these four values are equal to each other for every state variable in the model. Next, focusing on any action element reveals the action element's current properties, most of which are derived from the future event list: `isEnabled`, `timeToFire`, `isNextToFire`, `wasLastToFire`, and `numTimesFired`. Since the simulation batch is currently in the initial model state, and no actions have fired yet, the `wasLastToFire` property should be `false` for every action in the model. Also, note that all of the disabled actions, such as `UncoveredAttack`, `MissAttack`, and `DetectAttack`, show that `timeToFire` = -1.0. The invalid simulation time $t = -1.0$ is simply a placeholder for a simulation time that does not exist. The `timeToFire` value does not exist for any of the disabled actions since none of those action are currently represented by an event in the future event list.

Next, let us examine the usefulness of modifying the model state to explore rare events of the simulation. First, consider the scenario in which the dedicated IDS sensors are sparsely distributed in a relatively large AMI network. In this scenario, the likelihood that a single attack is covered by at least one IDS sensor is relatively low. In our model, this means that the `NoCoverage` action has a much higher rate than the `Coverage` action. In other words, the token in the `SharedAttack` state variable is much more likely to transfer directly to the `UndetectedAttack` state variable, causing the token to skip over the rest of the model entirely. Assuming we do not want to alter the rates of the `NoCoverage` and `Coverage` actions (since they could represent accurate parameters of the actual system), but we still want to examine the rest of the model simulation functionality for when these rare event do occur, we could simply modify the model state. To move the token from the `SharedAttack` state variable to the `Attack` state variable, we have two viable options. The first option is to simply remove the token from the `SharedAttack` state variable (`SharedAttack->Mark()=0`), and add a token to the `Attack` state variable (`Attack->Mark()=1`). That modification would cause the simulation to skip the race condition between the `NoCoverage` and `Coverage` actions, entirely. Another option is to alter the outcome of the race between the `NoCoverage` and `Coverage` actions by modifying the future event list of the running simulation. That modification is accomplished by editing the `timeToFire` value of the `Coverage` action to a simulation time that is earlier than the `timeToFire` value of the `NoCoverage` action. That modification will ensure that the token will move through the `Coverage` action to the `Attack` state variable, rather than through the `NoCoverage` action immediately to the final `UndetectedAttack` state variable.

```
(BinaryOperator AND
  (ActionBreakpoint OnFired 5) ;MissAttack->index=5
  (SimTimeBreakpoint 3.0)
)
```

Figure 4.4: Case Study Breakpoint Example.

Next, let us look at a use of breakpoints to run the simulation to a model state that conforms to some explicitly defined conditions. Consider the case in which a detectable attack, with an execution time of at least 3.0 simulation time units, goes undetected since all of the sensors covering the given attack location are currently unavailable. We can more specifically describe the case as the breakpoint in Figure 4.4. Applying the breakpoint to the running set of simulation batches allows us to pause a simulation batch on a model state conforming to the exact conditions specified in the breakpoint. From that particular model state, the analyst is free to explore the details of the model, such as the overall ratio of available sensors to unavailable sensors.

Another useful additional feature is the ability to step through the simulation, firing one action at a time. That feature is designed, in a sense, to bring the static model to life by showing each model state as the simulation batch progresses. In the case of the relatively small AMI model, a user could easily step through an entire simulation batch. Besides the oscillation of the set of tokens looping between the `BrokenSensors` and `WorkingSensors` state variables, the only token for the user to follow is the one initially residing in the `SharedAttack` state variable. That token has only three possible paths and two possible destinations, as pictured in Figure 4.5. Following the token allows the analyst to examine each model state of the simulation batch at each point along its current path. Consider, now, the case in which the modeler is searching for a set of model parameters, such as the ratio of the `MissAttack` rate to the `DetectAttack` rate, to create an interesting set of system results. Simulation stepping, together with model state modification, provides the user with the ability to easily, and quickly, tweak these parameters to find the desired values.

4.2 SAN with SAN Debugger

Extending the inherited functionality of the AFI debug and visualization editor, the SAN debug and visualization editor also provides a powerful graphical visualization of the model state. As discussed in the SAN subsection of Section 3.3.2, these visualizations include associating the intensity of the color of a `Place` visual element with the number of tokens

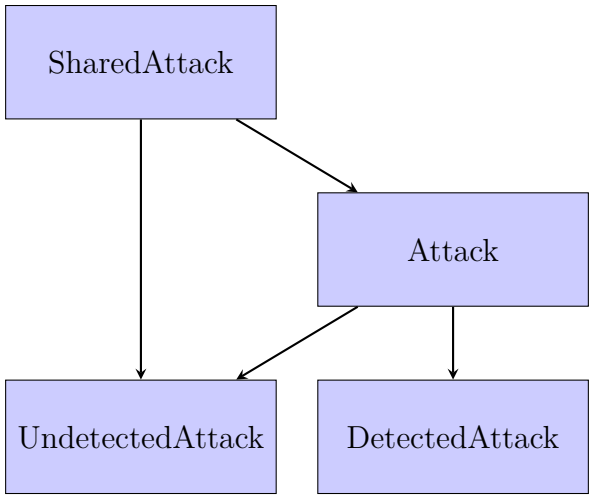


Figure 4.5: Possible Paths of the SharedAttack Token.

currently in that `Place`, and highlighting `Activity` visual elements with different colors indicating the value of its `wasLastToFire` and `isNextToFire` fields. These simple additions to the debug and visualization editor allow the user to quickly comprehend the most important aspects of the current model state without having to access the specific properties of each model element, as is the case with the AFI debug and visualization editor.

Continuing with the AMI case study, the initial model state is pictured in Figure 4.6. From the visualization alone, it is obvious that the `SharedAttack` token is currently in the `SharedAttack` place, while the other places along the flow—`Attack`, `UndetectedAttack`, and `DetectedAttack`—are empty. It is also fairly intuitive to determine the ratio between the number of available and unavailable sensors from the color intensities of the `WorkingSensors` and `BrokenSensors` place elements. Also, it is easy to note that the next activity to fire is `Coverage`, since it is highlighted in green. Stepping one activity forward in the simulation batch results in the model state pictured in Figure 4.7. In addition to all of the information from the previous state, the user can also visually determine the last activity that fired (`Coverage`), since it is highlighted in red.

4.3 Composed Model

To demonstrate the functionality of the Rep/Join debug and visualization editor, the model of the AMI case study is expanded to the more specific representation presented in Figures 4.8–4.11. In this new representation, each AMI meter is represented as being covered by 0, 1, or 2 IDS sensors, to distinguish between the different levels of security between AMI

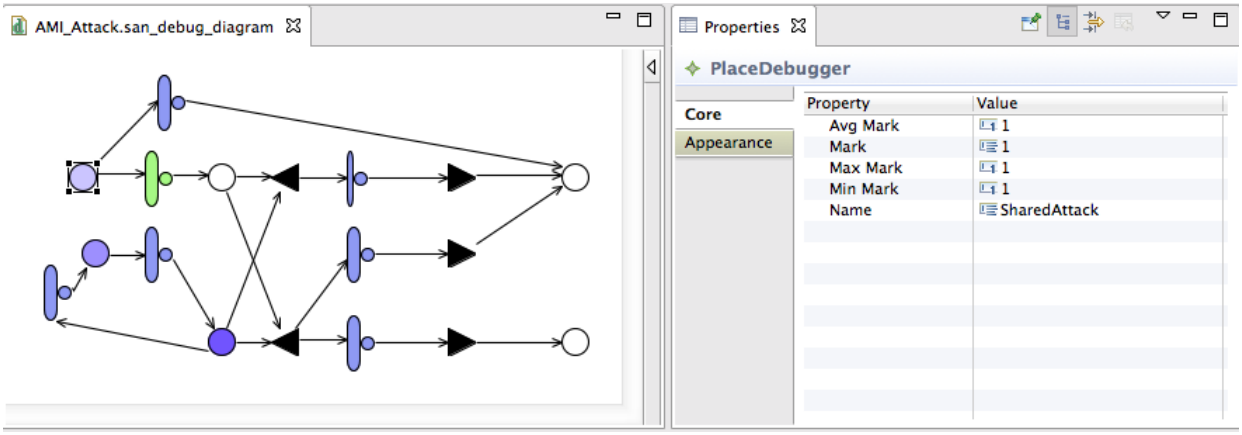


Figure 4.6: Initial State of SAN Model Using SAN Debug and Visualization Editor.

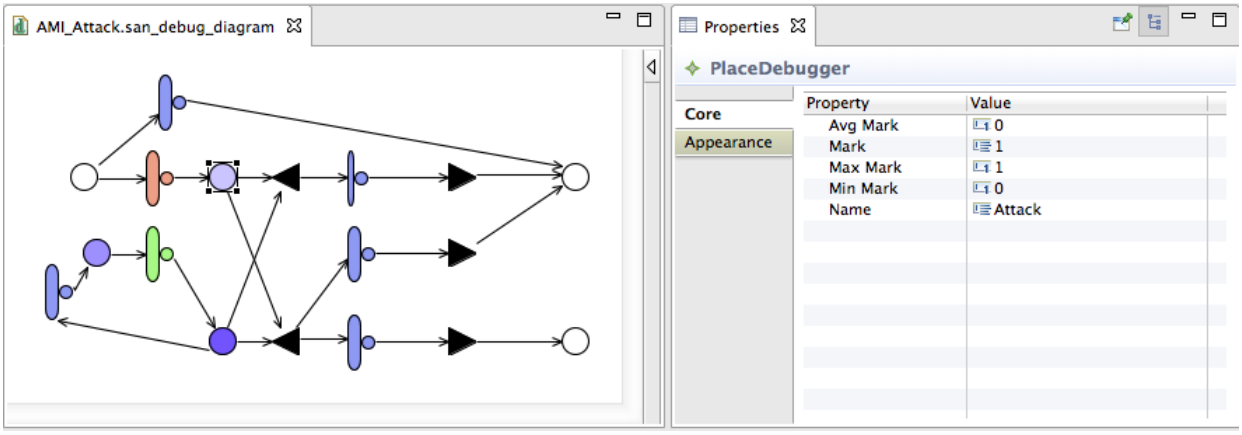


Figure 4.7: Second State of SAN Model Using SAN Debug and Visualization Editor.

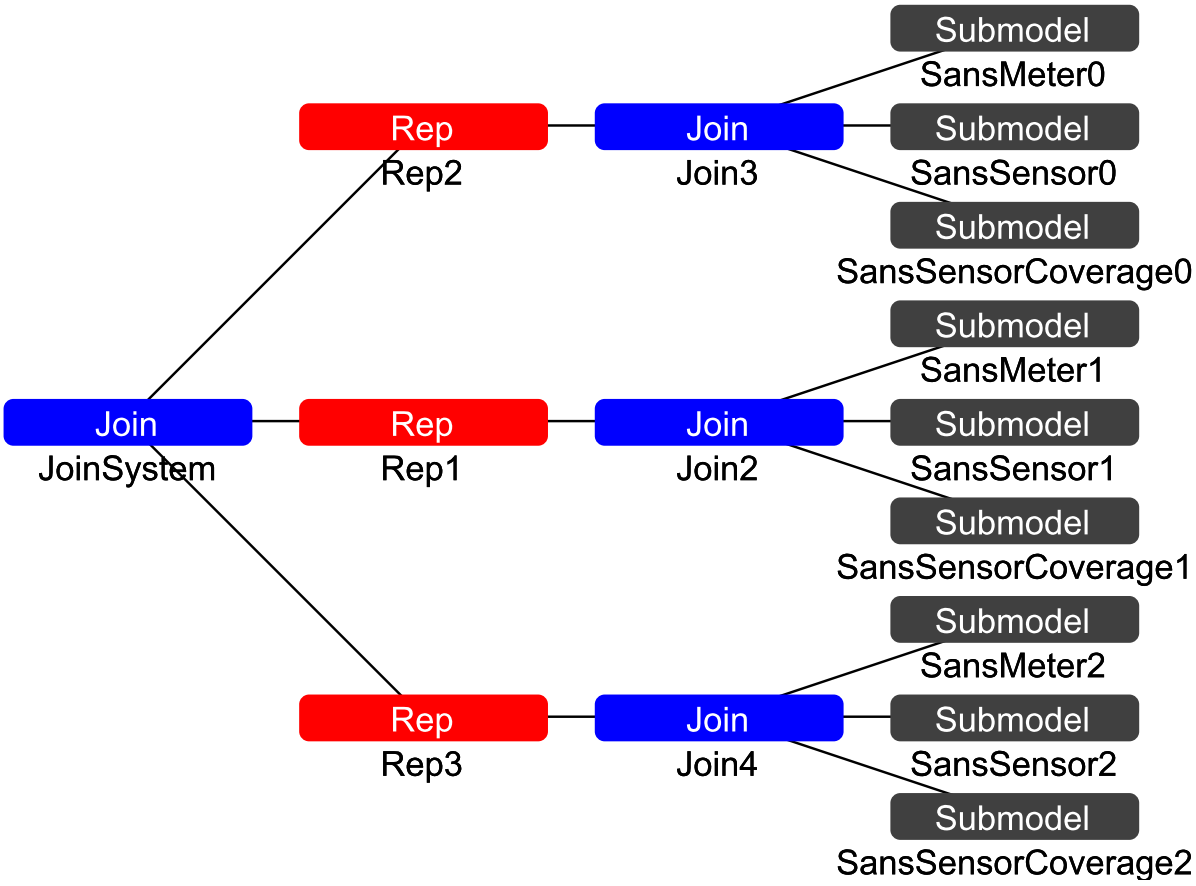


Figure 4.8: Comp: Single Attack on AMI with Dedicated IDS Architecture (Overall Model).

meters that have differing amounts of coverage. For example, consider the case where an IDS sensor has an availability of 50%. If an AMI meter is covered by 0 sensors, it is covered 0% of the time. If it is covered by 1 sensor, it is covered roughly 50% of the time. If it is covered by 2 sensors, it is covered roughly 75% of the time. This added distinction from the previous, and simpler, case study model allows us to also consider IDS sensor deployment within varying densities of AMI meter locations. The important thing to notice here, for our discussion, is that the Rep/Join model, in Figure 4.8, is composed of the atomic SAN models, in Figures 4.9–4.11.

As with all modeling formalisms in the Möbius modeling framework, a simulation batch of this model can be represented by the AFI debug and visualization editor. However, as noted in Section 1.5.3, one of the goals of the composed model implementation is to retain as much scalability as possible. As the AFI version of this model would result in upwards of 60 state variables and 45 actions (much more with Rep instances > 1), a specific Rep/Join debug and visualization editor must be employed.

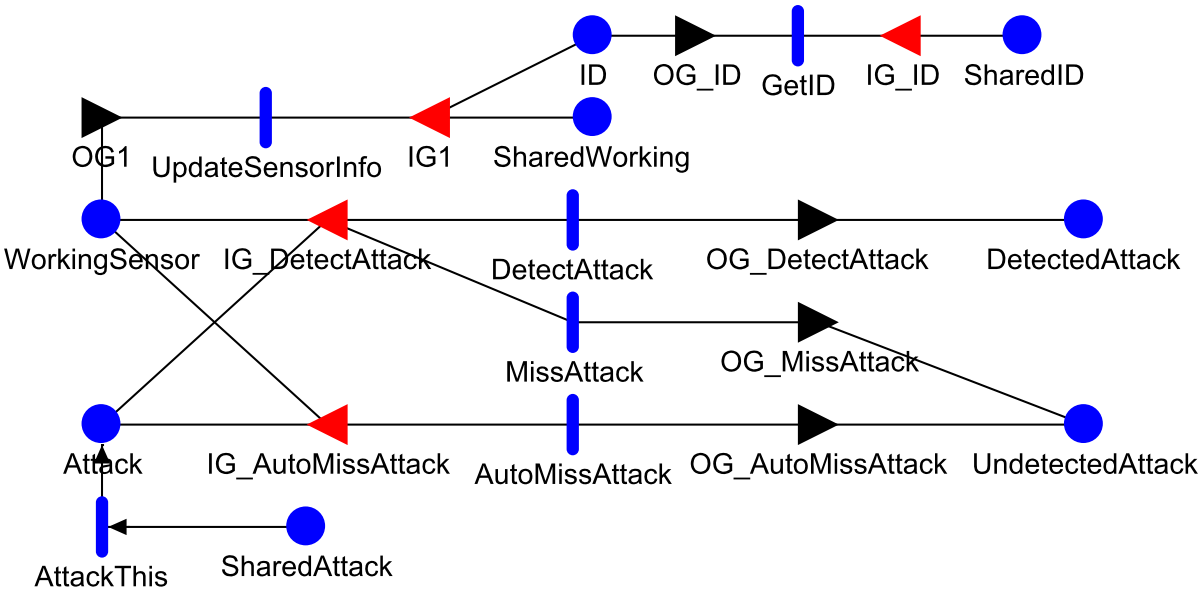


Figure 4.9: Comp: Single Attack on AMI with Dedicated IDS Architecture (SansMeter).

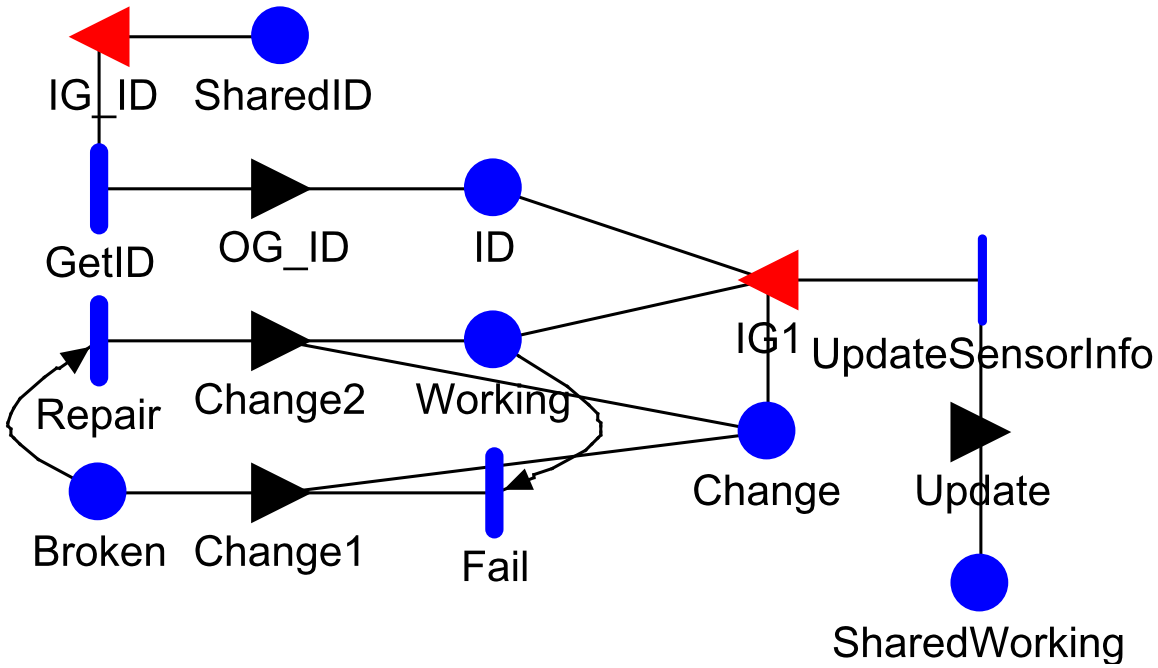


Figure 4.10: Comp: Single Attack on AMI with Dedicated IDS Architecture (SansSensor).

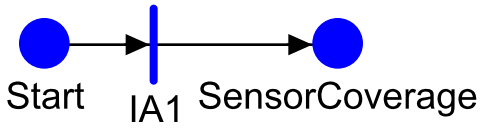


Figure 4.11: Comp: Single Attack on AMI with Dedicated IDS Architecture (SansSensorCoverage).

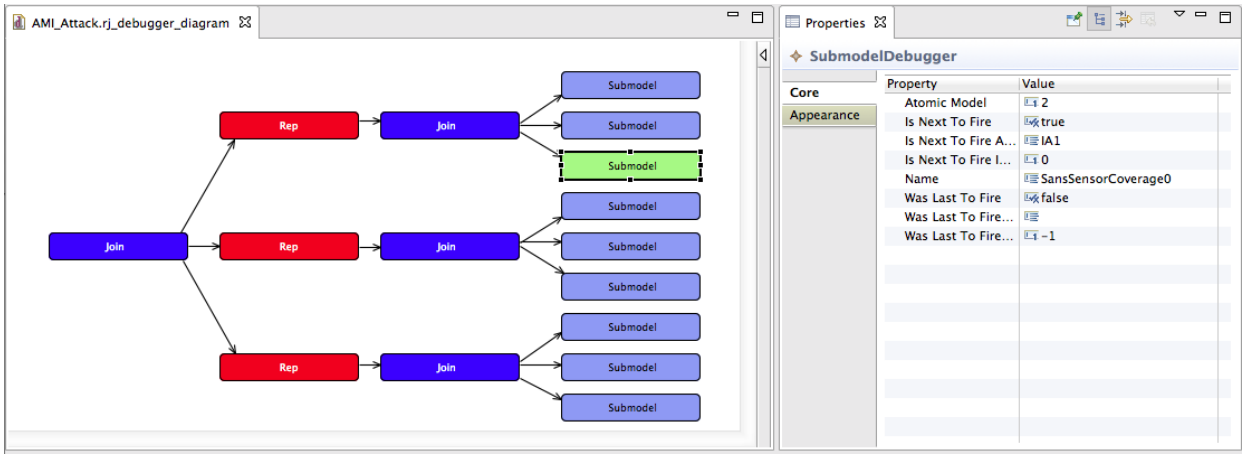


Figure 4.12: Initial State of Rep/Join Model Using Rep/Join Debug and Visualization Editor.

First, let us explore the initial model state using the Rep/Join debug and visualization editor. In this editor, we have a clear view of the overall model state, as shown in Figure 4.12. Specifically, we can view and modify the shared state variables of the `Rep` and `Join` elements of the model, and we can easily identify the next action to fire. Taking one step into the simulation, we can also easily determine the last action that has fired, as shown in Figure 4.13. In addition to stepping through the simulation, we can also specify breakpoints in this editor using conditions from specific `Submodel` elements. Using this editor, we can effectively analyze large Rep/Join composed models.

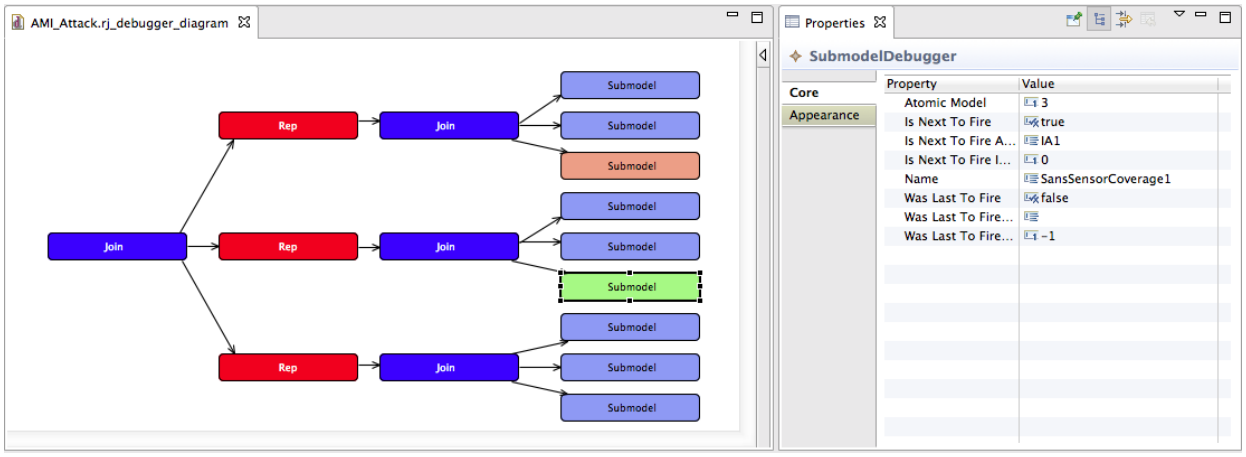


Figure 4.13: Second State of Rep/Join Model Using Rep/Join Debug and Visualization Editor.

CHAPTER 5

CONCLUSIONS AND FUTURE RESEARCH

In this thesis, we first discussed the benefits of applying debugging and visualization capabilities to running discrete-event simulations, instead of simply applying them to the original model or to the final results of the discrete-event simulation. Next, we defined the features that would help us accomplish our debugging and visualization goals most effectively. We then described how we implemented each of the discussed features and goals in the Möbius discrete-event simulator. We then examined the utility of our work in light of a real-world analysis problem: distribution considerations of an IDS in an AMI.

Although this work provides the foundation of a powerful tool for the debugging and visualization of running discrete-event simulations, some ideas either required unavailable resources or fell out of the scope of this specific project. Two of those ideas include integration with the GNU Project debugger (GDB) for complex models, and a more interactive debug and visualization editor for composed models.

Some models contain complex user-defined code, such as in input gate predicates in SAN models. Currently, there is no way for end users to debug those code fragments with a debugging tool. As the current implementation of the MSDV tool presented in this paper only addresses the model state, individual and direct analysis of such code fragments is still not possible. One solution would be to integrate the MSDV tool with GDB to allow users to debug the code through the Möbius modeling tool.

Another possible improvement would address the visibility versus mutability issue present in the current implementation of the debug and visualization editor for composed models. In the current implementation, the user cannot access or modify the future event list or the unshared state variables of specific submodels. One possible solution would be to implement a drill-down functionality through which the user can inspect specific submodels through the visualization implementation specific to the modeling formalism of that submodel. That solution would offer the user more control over the granularity of the visualization and, hence, more control over the specifics of the simulation batch.

APPENDIX A

IMPLEMENTING NEW VISUALIZATION USER INTERFACES IN MSDV

This appendix contains a concise guide on how to implement new visualization user interfaces in the Möbius Simulation Debugger and Visualization (MSDV) feature. For the full, up-to-date guide on this topic, visit the Möbius Wiki [39].

A.1 Setup and Preliminary Assumptions

Throughout this guide, the term *developer* will refer to the one(s) who is (are) developing the new debug and visualization interface for MSDV. The term *user* will refer to the end user who uses the final features available in the Möbius tool, including the additional features provided by the aforementioned developer.

This guide assumes the developer has a successfully running version of the Möbius source code. The steps to accomplish that preliminary task can also be found on the Möbius Wiki [39].

Also, this guide assumes the developer has an adequate background in the following technologies: Eclipse RCP development [40], Eclipse Modeling Framework (EMF) development [41], and Eclipse Graphical Modeling Framework (GMF) development [42]. These technologies can be explored through the respective documentation of each technology.

A.2 Defining the Debug and Visualization Interface

After completing the preliminary setup of the Möbius project, the first step of the developer is to define the properties of the new debug and visualization interface. More specifically, the developer must define which properties are to be accessible to the user. These developer-defined properties may include simple statistics, such as the `minMark` and `maxMark` properties discussed in Section 3.3.1, or any other statistics that the developer may believe to be helpful to the user. To define these properties, the developer must create a new "Empty EMF Project" in the Möbius Eclipse workspace. In the newly created project, the developer must

create a new "Ecore Model" and name it with the standard workspace naming conventions[1]. In the model, the developer must select the "Load Resource..." option from the main element dropdown list, and select the model editor *.ecore file of the modeling formalism to extend. Next, the developer must create a `*ModelDebugger` object as the root of the newly created interface model. Then, each displayable element in the new debug and visualization interface must have an associated model `*Debugger` object that inherits from its respective model object in its modeling formalism model editing model. Next, the developer must create EReferences from the `*ModelDebugger` root object to each of the other `*Debugger` objects in the model, all of which as containment references with the range of $[0, *]$. From here, the developer can now add the additional debug and visualization properties to each of the `*Debugger` objects.

To examine the previous steps with a concrete example, we describe the steps as they relate to the creation of the SAN debug and visualization editor, originally presented in Section 3.3.2. In this example, each of the SAN elements of the SAN model editor (Figure 3.10) are inherited by the elements of the SAN debug and visualization editor (Figure 3.12). Each of the SAN debug and visualization editor elements now have access to both its parent properties, such as `mark` for the `Place` element, and its additional debug and visualization properties, such as `minMark` for the `Place` element. All of these properties are now accessible to the debug and visualization editor to be either displayed directly to the user, or to be used to generate an appropriate visualization of the current state of the model during the simulation.

After creating the Ecore model of the debug and visualization editor, the developer must define the default visual presentation of each of the elements in the model using the associated GMF files. This step should be familiar to developers with adequate GMF experience.

A.3 Integrating the Debug and Visualization Interface into MSDV

Much of the integration of the newly created debug and visualization interface happens automatically since MSDV already checks for the debug and visualization interface for the specific modeling formalism. Now that the interface has been created, MSDV no longer defaults to the AFI implementation of the debug and visualization interface when attempting to access a model defined with the specific modeling formalism.

[1] At the time of this writing, the standard naming convention for the Ecore model file is the same as the name of the model of its respective modeling formalism editor concatenated with the "_debug" suffix. See other projects in the workspace to conform to the current naming convention of the workspace.

To complete the integration, however, the developer must define each of the debug and visualization properties of the debug and visualization Ecore model through Java code. For example, the user must implement the meaning of `minMark` through its Java function implementation. This functionality is added to the `UpdateInterface` function, the function that is most notably called from the Möbius `StreamReader` class when the back-end Möbius simulation processes forward messages to the front-end Java interface. The `UpdateInterface` function allows the developer to update the values of the properties in the debug and visualization editor, and, also, to update the visual presentation of the model elements in the debug and visualization editor.

REFERENCES

[1] T. Jazouli, P. Sandborn, and A. Kashani-Pour, "A Direct Method for Determining Design and Support Parameters to Meet an Availability Requirement," *International Journal of Performability Engineering*, vol. 10, no. 2, pp. 211–225, Mar. 2014.

[2] J. Andersson, "Environmental Impact Assessment using Production Flow Simulation," Chalmers University of Technology, Department of Product and Production Development, Gothenburg, Sweden, Tech. Rep. 85, 2014.

[3] J. Viana, S. C. Brailsford, V. Harindra, and P. R. Harper, "Combining Discrete-event Simulation and System Dynamics in a Healthcare Setting: A Composite Model for Chlamydia Infection," *European Journal of Operational Research*, Mar. 2014.

[4] D. Goldsman, R. E. Nance, and J. R. Wilson, "A Brief History of Simulation Revisited," in *Proceedings of the 2010 Winter Simulation Conference, ed., B. Johansson, S. Jain, J. Montoya-Torres, J. Hugan, and E. Ycesan*, Piscataway, New Jersey, 2010, pp. 567–574.

[5] J. J. Swain, "To Boldly Go...Discrete Event Simulation Software Tools," *OR/MS Today*, vol. 35, no. 5, pp. 50–61, Oct. 2009.

[6] G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. Webster, "The Möbius Modeling Tool," in *Proceedings of the 9th International Workshop on Petri Nets and Performance Models*, Aachen, Germany, Sep. 11–14, 2001, pp. 241–250.

[7] A. J. Stillman, "Model Composition within the Möbius Modeling Framework," M.S. Thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 1999.

[8] "Simul8 Website," 2014. [Online]. Available: http://www.simul8.com/

[9] "Vensim Website," 2014. [Online]. Available: http://vensim.com

[10] A. L. Williamson, "Discrete Event Simulation in the Möbius Modeling Framework," M.S. Thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 1998.

[11] A. Kuratti, "Improved Techniques for Parallel Discrete Event Simulation," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Urbana, IL, 1997.

[12] J. Peccoud, T. Courtney, and W. H. Sanders, "Möbius: An Integrated Discrete-Event Modeling Environment," *Bioinformatics*, vol. 23, no. 24, pp. 3412–3414, 2007.

[13] S. Derisavi, "The Möbius State-Level Abstract Functional Interface," M.S. Thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 2003.

[14] J. M. Doyle, "Abstract Model Specification using the Möbius Modeling Tool," M.S. Thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 2000.

[15] S. Derisavi, P. Kemper, W. H. Sanders, and T. Courtney, "The Möbius State-level Abstract Functional Interface," *Performance Evaluation*, vol. 54, no. 2, pp. 105–128, Oct. 2003.

[16] D. D. Deavours, "Formal Specification of the Möbius Modeling Framework," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Urbana, IL, 2001.

[17] W. H. Sanders and J. F. Meyer, "Stochastic Activity Networks: Formal Definitions and Concepts," in *Lectures on Formal Methods and Performance Analysis, First EEF/Euro Summer School on Trends in Computer Science*, Berg en Dal, The Netherlands, July 3–7, 2000, pp. 315–343.

[18] L. M. Malhis, "Development and Application of an Efficient Method for the Solution of Stochastic Activity Networks with Deterministic Activities," Ph.D. dissertation, University of Arizona, Tuscon, AZ, 1996.

[19] W. H. Sanders, "Construction and Solution of Performability Models Based on Stochastic Activity Networks," Ph.D. dissertation, University of Michigan, Ann Arbor, MI, 1988.

[20] E. A. Lemay, "Adversary-Driven State-Based System Security Evaluation," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Urbana, IL, 2011.

[21] M. D. Ford, "A Generalized Adversary Decision Algorithm and Analytic Solution Methods for ADVISE Models," M.S. Thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 2012.

[22] D. C. Eskins, "Modeling Human Decision Points in Complex Systems," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Urbana, IL, 2012.

[23] W. D. Obal, "Importance Sampling Simulation of SAN-Based Reward Models," M.S. Thesis, University of Arizona, Tucson, AZ, 1993.

[24] T. Courtney, S. Gaonkar, M. Griffith, V. Lam, M. McQuinn, E. Rozier, and W. H. Sanders, "Data Analysis and Visualization within the Möbius Modeling Environment," in *Proceedings of the 3rd International Conference on the Quantitative Evaluation of SysTems (QEST)*, Riverside, California, Sep. 11–14, 2006, pp. 137–138.

[25] R. Lamprecht and P. Kemper, "Möbius Trace Analysis with Traviando," in *Proceedings of the 6th International Conference on the Quantitative Evaluation of SysTems (QEST 2008)*, St. Malo, France, Sep. 2008, pp. 41–42.

[26] P. Kemper and C. Tepper, "Automated trace analysis of discrete event systems models," *IEEE Transactions on Software Engineering*, vol. 35, no. 2, pp. 195–208, 2009.

[27] S. Gaonkar, K. Keefe, R. Lamprecht, E. Rozier, P. Kemper, and W. H. Sanders, "Performance and Dependability Modeling with Möbius," *ACM Performance Evaluation Review*, vol. 36, no. 4, Mar. 2009.

[28] S. K. Klock and P. Kemper, "An Automated Technique to Support the Verification and Validation of Simulation Models," in *Proceedings of Dependable Systems and Networks (DSN 2010)*, Chicago, Illinois, June 2010.

[29] P. Kemper and C. Tepper, "Traviando - A Trace Analyzer to Debug Simulation Models," in *Proceedings of the 19th Symposium on Simulation Technique (ASIM 2006)*, Hannover, Germany, Sep. 12–14, 2006.

[30] P. Kemper and C. Tepper, "Traviando - Debugging Simulation Traces with Message Sequence Charts," in *Proceedings of the 3rd International Conference on the Quantitative Evaluation of SysTems (QEST 2006)*, Riverside, California, Sep. 11–14, 2006, pp. 135–136.

[31] W. H. Sanders and L. M. Malhis, "Dependability Evaluation Using Composed SAN-Based Reward Models," *Journal of Parallel and Distributed Computing, Special Issue on Petri Net Models of Parallel and Distributed Computers*, vol. 15, no. 3, pp. 238–254, July 1992.

[32] W. H. Sanders and J. F. Meyer, "Reduced Base Model Construction Methods for Stochastic Activity Networks," in *Proceedings of the Third International Workshop on Petri Nets and Performance Models*, Kyoto, Japan, Dec. 11–13, 1989, pp. 74–84.

[33] W. H. Sanders and R. S. Freire, "Efficient Simulation of Heirarchical Stochastic Activity Network Models," *Discrete Event Dynamic Systems: Theory and Applications*, vol. 3, no. 2/3, pp. 271–300, July 1993.

[34] D. Grochocki, J. H. Huh, R. Berthier, R. Bobba, W. H. Sanders, A. A. Cárdenas, and J. G. Jetcheva, "AMI Threats, Intrusion Detection Requirements and Deployment Recommendations," in *Proceedings of the 3rd IEEE International Conference on Smart Grid Communications (SmartGridComm)*, Tainan City, Taiwan, Nov. 5–8, 2012, pp. 395–400.

[35] D. R. Grochocki, "Deployment Considerations for Intrusion Detection Systems in Advanced Metering Infrastructure," M.S. Thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 2013.

[36] R. Berthier, W. H. Sanders, and H. Khurana, "Intrusion Detection for Advanced Metering Infrastructures: Requirements and Architectural Directions," in *Proceedings of the 1st IEEE International Conference on Smart Grid Communications (SmartGridComm)*, Gaithersburg, Maryland, Oct. 4–6, 2010, pp. 350–355.

[37] R. Berthier and W. H. Sanders, "Specification-based Intrusion Detection for Advanced Metering Infrastructures," in *Proceedings of the 17th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2011)*, Pasadena, California, Dec. 12–14, 2011, pp. 184–193.

[38] R. Berthier, J. G. Jetcheva, D. Mashima, J. H. Huh, D. Grochocki, R. Bobba, A. A. Cárdenas, and W. H. Sanders, "Reconciling Security Protection and Monitoring Requirements in Advanced Metering Infrastructures," in *Proceedings of the IEEE International Conference on Smart Grid Communications (SmartGridComm)*, Vancouver, Canada, Oct. 21–24, 2013.

[39] "Möbius Wiki," 2014. [Online]. Available: https://www.perform.illinois.edu/wiki

[40] "Eclipse Rich Client Platform," 2014. [Online]. Available: http://wiki.eclipse.org/Rich_Client_Platform

[41] "Eclipse Modeling Framework Project," 2014. [Online]. Available: https://www.eclipse.org/modeling/emf/

[42] "Graphical Modeling Framework," 2014. [Online]. Available: http://wiki.eclipse.org/Graphical_Modeling_Framework