DESIGN AND IMPLEMENTATION OF AN EXTENSIBLE TOOL FOR PERFORMANCE AND DEPENDABILITY MODEL EVALUATION

BY

GERARD PATRICK KAVANAUGH III

B.S., Virginia Polytechnic Institute and State University, 1995

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1998

Urbana, Illinois

ABSTRACT

The goal of the Möbius project is to develop an object-oriented, formalism-independent, stochastic modeling framework, and to implement the framework in a practical, performance-dependability evaluation tool. The Möbius framework is the specification of an abstract notion of modeling in which models may be represented in many formalisms. The Möbius tool uses this flexible modeling approach for model construction, and implements a means for solvers to interact with these models in order to determine measures of interest. More than simply providing an interface for the framework, the Möbius tool is the realization of architectural concepts that allow it to be both extensible and maintainable.

This work describes the architecture of the Möbius tool, as well as the features it provides for the future inclusion of new functionality. Specifically, the tool's main application is developed, as well as the implementation of extensible base classes from which the tool's functional units or modules for model construction and solution may be derived. The types of modules supported are also defined. In particular, these consist of atomic model constructors, composed model constructors, reward model constructors, studies, and solvers. In addition, the Möbius tool provides a means for the modular construction and solution of models through these modules, and was developed such that new modules will be simple to build and include in the future. Further, the implementation of the tool shows both the usefulness of our approach and the flexibility of the Möbius modeling framework.

To all my loved ones

ACKNOWLEDGMENTS

I would like to begin by thanking Dr. William Sanders for his confidence in me and for his technical guidance with my work in the PERFORM Group. Further, I would like to thank John Sowder, Alex Williamson, and Jay Doyle for their friendship and dedication on the Möbius project. Thanks also to Dan Deavours and Doug Obal for their technical expertise and advice throughout my research. Specials thanks also to Jenny Applequist for her help in reviewing this thesis.

Additional thanks go to the Defense Advanced Research Projects Agency, Information Technology Office, for funding under contract DABT63-96-C-0069, and to the Motorola Space Systems Technology Group, including Renee Langefels and Peter Alejandro, for their long-term funding of the *UltraSAN* and Möbius projects.

Finally, I would like to thank my family for their constant support, Shannon for her incredible devotion and her constant inspiration, and all my friends for just being who they are. Lastly, I would like to thank God for giving me more than I ever could have asked for.

TABLE OF CONTENTS

Page

1. INTRODUCTION....1
1.1. Modeling Overview 1
1.2. Möbius Framework Introduction 2
1.3. Research Objectives....4
2. ARCHITECTURAL OVERVIEW 6
2.1. Our Basic Architecture....6
2.2. Overview of the Möbius Tool....7
2.2.1. Module access....8
2.2.2. Module overview 9
2.2.3. Rapid application design....11
2.2.4. Implementation overview 12
3. THE CONTROL PANEL 17
3.1. Introduction....17
3.2. Initialization 18
3.3. Module and Model Access....19
3.3.1. Creating and opening models....19
3.3.2. Adding and removing modules....24
3.4. Supporting Classes....25
3.4.1. Utility classes 25
3.4.2. Dialog windows 26
4. CLASSES TO SUPPORT MODULE IMPLEMENTATION 28
4.1. Module Organization 28
4.2. Interface Classes 30
4.2.1. Dependency maintenance....30
4.2.2. Saving 32
4.2.3. Validating models 35
4.2.4. Propagation of interface saves 36
4.2.5. Loading module interfaces for access 40
4.2.6. Compiling 41
4.2.7. Importing....41
4.3. Editor Classes....42
4.3.1. Undo functionality 42
4.3.2. File menu commands 43
4.3.3. Textual editors 46
4.3.4. Graphical editors 46
4.3.5. Global variables 56
4.4. Information Classes....57
4.4.1. Compilation 57
4.4.2. Communication....58
5. CONCLUSIONS AND FUTURE RESEARCH....60

APPENDIX: MÖBIUS TOOL FILE STRUCTURE 62
REFERENCES 63

LIST OF FIGURES

Figure Page

Figure 1: Möbius Framework Formalism Types 3
Figure 2: Basic Architecture Structure....7
Figure 3: Module Relationship Diagram 11
Figure 4: Möbius Tool Architecture 12
Figure 5: Model Transformation Process....14
Figure 6: Model Transformation Implementation....15
Figure 7: The Möbius Control Panel 17
Figure 8: Control Panel Model Access 20
Figure 9: Type and Class Lists....21
Figure 10: Module Composition....28
Figure 11: Dependency Graph Example 31
Figure 12: Dependency Node Information Example 32
Figure 13: Interface Save Functionality Flow Chart Diagram 33
Figure 14: Example of a Propagation List 37
Figure 15: Propagation List Formation 38
Figure 16: Möbius Simulator Textual Editor Example....46
Figure 17: Möbius SAN Editor Graphical Editor Example 48
Figure 18: SAN Editor Panel Menu 49
Figure 19: SAN Editor Edge Examples 55
Figure 20: Möbius Tool File Structure 62

1. INTRODUCTION

1.1. Modeling Overview

The use of modeling to predict system behavior is useful during system conception, design, and analysis. Many important system behaviors rarely occur, and many systems are too costly to allow the building of prototypes of many design alternatives. For these and similar reasons, computer modeling is becoming increasingly popular in many fields. Since the early 1950s, people have been applying queuing network theory to systems of interest in order to determine performance characteristics, such as average customer waiting time and average queue length.

The process of modeling leaves the modeler with a large number of possible decisions, such as which formalism to use, how to solve the model, and how to specify the performance variables of interest. Today, although queuing theory is still in use, there are many formalisms that can be used to specify complex system behavior, including, for example, Petri nets [1] and stochastic activity networks [2]. These formalisms allow modelers to specify systems with contention for resources and state-dependent delay times. Furthermore, as computers have grown increasingly faster, modelers have been able to simulate larger systems, and analytically solve larger Markov models.

There have thus been many tools designed to facilitate the specification and solution of models. Some examples of this type of tool are *UltraSAN* [3], *HiQPN* [4], and *SHARPE* [5]. Although all of these tools have high-level model interfaces that allow for rapid model specification, the modeler is still often restricted by the means of either model specification, or solution, or both. Some formalisms are very useful for representing certain types of system, or even parts of larger systems, while others may be superior for different systems. For example, stochastic activity networks are easier to use for modeling resource contention than queuing networks are. Some tools, such as SHARPE, allow model specification in multiple formalisms (with the restriction that information is passed by results), whereas most tools permit the use of only a single formalism. Although some tools, for example *UltraSAN*, support multiple means for solution, integrating new solvers is often a complicated task.

From a research standpoint, it is difficult to expand these tools to accommodate new research. Consequently, research based on these tools may be very limited by their capabilities.

The Möbius tool's architecture, presented in this thesis, implements a means for model construction that is described by the Möbius framework [6]. The tool's model construction methods not only allow the specification of models in multiple formalisms, but also allows formalisms to share notions of state between them. This enables system behavior to be specified in whatever formalism is most convenient. Further, the tool permits researchers to apply existing solvers to new formalisms. In this way, new formalisms or means for model specification do not mandate the implementation of special solvers. The tool also allows modelers to rapidly design interfaces for specifying models in these formalisms. In addition, the Möbius tool permits large models to be composed of smaller, simpler models, which may be reused in multiple systems, thereby making a modeler's job easier. Similar to code reuse, this allows modelers to maintain libraries of components that may be composed to define larger models quickly and easily.

1.2. Möbius Framework Introduction

The Möbius framework incorporates a notion of modeling that is less restrictive than in existing tools. It is a framework in which new formalisms, new model construction and solution techniques, and new means of reward specification may be incorporated easily. Different notions of state in different formalisms may be shared, and solvers may be designed to analyze models specified in formalisms that have yet to be written. The framework is rigorously defined in [6]. An overview is presented here, since a general understanding of the framework is required in order to understand the work presented in this thesis.

The Möbius framework, as shown in Figure 1, uses several different types of "formalisms" during model specification. In this context, *formalisms* are high-level modeling languages used for system representation. One or more of these formalism types are used to define a model within the framework. Each different formalism type plays a different and important role in the construction of the larger model. *Actions* in the Möbius framework define the components in a model that change its "state." *State variables* within the Möbius framework are the basic units that have values, and are acted upon by the actions. The

combination of the values of the state variables in a model represents the *state* of the model. Further, “models” in each formalism are built using state variables, actions, and/or other “models.” A *model* in the framework is a behavioral description of a system or portion of a system expressed in a particular formalism.

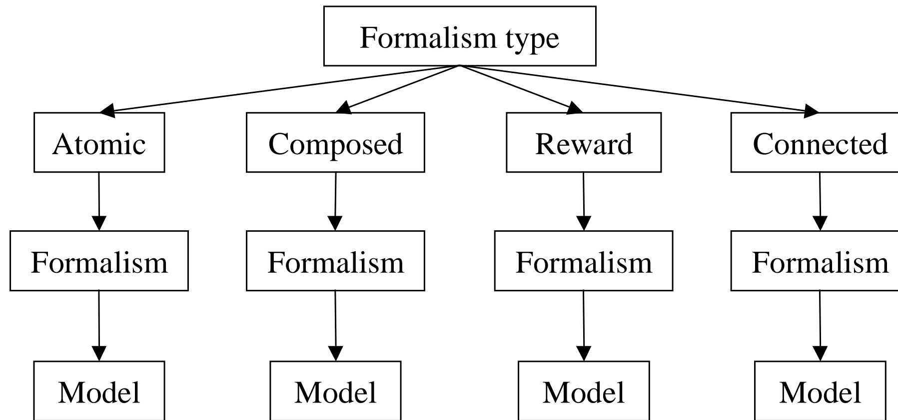

Figure 1: Möbius Framework Formalism Types

The following provides a brief description of each formalism type:

- **Atomic Model**: An atomic model is defined by a set of actions, state variables, and the rules describing the interactions between the two.
- **Composed Model**: A composed model specifies a function that takes one or more models (atomic or composed) as its input and returns a model as its output. Certain composition functions are able to exploit symmetries to reduce state-space size and specification requirements for large models. Examples of composed models include the SAN Replicate/Join formalism [2] and composed model graphs [7].
- **Reward Model**: A reward model consists of a model and a set of measures that are defined by functions of state on that model. Reward models may also augment the state of the underlying model. In order for numerical solvers to work with these reward models, the entire reward must be encoded in the state. For example, some performance measures require that the knowledge of the last action completed be added to the underlying model’s state.

- **Connected Model**: A connected model consists of a collection of models that are dependent in such a way that the measures (results) of one model are passed as parameters to another.

Although the formalism types in the Möbius framework have been described, no description of specific types of state variables, actions, compositions, or performance variables was necessary to do so. This is precisely the goal of the framework. Different formalisms may have completely different notions of state or action. Rather than specifying what these notions are, the framework approaches modeling from a different direction, allowing many different notions of these abstract concepts. Because of this, a system's behavior may be specified in terms of whatever formalism is most useful, or its behavior may be decomposed into different operating modes that may be modeled separately.

1.3. Research Objectives

Although the Möbius framework provides a flexible and abstract approach to modeling, it is not a tool. It is a theoretical means for mathematically specifying large models in terms of submodels of one or more formalisms. This thesis describes the development of an approach (reflected in what we call an "extensible architecture") for developing applications such as the Möbius tool, and then shows how that architecture was used to develop the Möbius tool itself.

More specifically, this thesis will

- Define an architectural approach for the development of extensible and maintainable applications like the Möbius tool.
- Describe the implementation of the control panel, which acts as the Möbius tool's main application.
- Describe the implementation of the interface, editor, and information classes that are the base classes for any model specification, composition, reward specification, and solution "module" in the tool.
- Provide a means for rapidly implementing new interfaces in the implementation.
- Provide a system into which new research may be easily integrated.

- Provide a system for defining models such that they may be reused and recomposed in a variety of ways.

The remainder of this thesis is organized as follows. Chapter 2 provides a description of the development of the extensible architecture, including the types of functions it must support and the general concepts that define it. It also provides an overview of how the Möbius tool was implemented, using the architecture as a guide. Chapter 3 describes the implementation of the control panel, which is the main application of the tool. Chapter 4 discusses the three main components of a module (an interface in the tool for model construction or solution), and describes their implementation. These components are the interface, editor, and information classes respectively. Chapter 5 summarizes the results of this thesis, describes the conclusions drawn from this work, and suggests areas of future work that are both implementation- and research-oriented.

2. ARCHITECTURAL OVERVIEW

This chapter presents an abstract notion of the architecture used in building the Möbius tool. As much as possible, discussion of the actual implementation of the tool will be left for later chapters. The goal of this chapter is to give the reader a broad understanding of the functionality that the tool requires. Later chapters will provide details on how this functionality is achieved.

2.1. Our Basic Architecture

In defining an architectural approach to support development of the Möbius performance/dependability evaluation tool, we aimed for an approach that would be extensible, maintainable, user-friendly, and useful. By extensibility, we mean that modules may be added to or removed from the application. By maintainability, we mean that the application should be designed such that modifications are possible. All applications contain implementation errors, and in order to fix these errors, the application's support staff will have to alter the implementation of the main application, the modules, or both. Correcting implementation errors or adding additional features should not invalidate the previous version's data. In addition, the application should be user-friendly in the sense that modules should be similar in their interfaces. Lastly, modules should be able to share information among themselves and users should be able to record previous progress or specifications within the application's various modules to ensure that the application is useful.

Figure 2 shows the structure that was developed for the architecture in order to meet these goals. It has a central main application from which the modules may be launched. It also has a well-defined, although modifiable, set of module types. A *module type* is a functional classification for a group of modules in the tool. It is important to define these module types so that the tool may be well-organized. By specifying module types through which access to individual modules is possible, the application designer is forced to choose the types of function that the tool will allow.

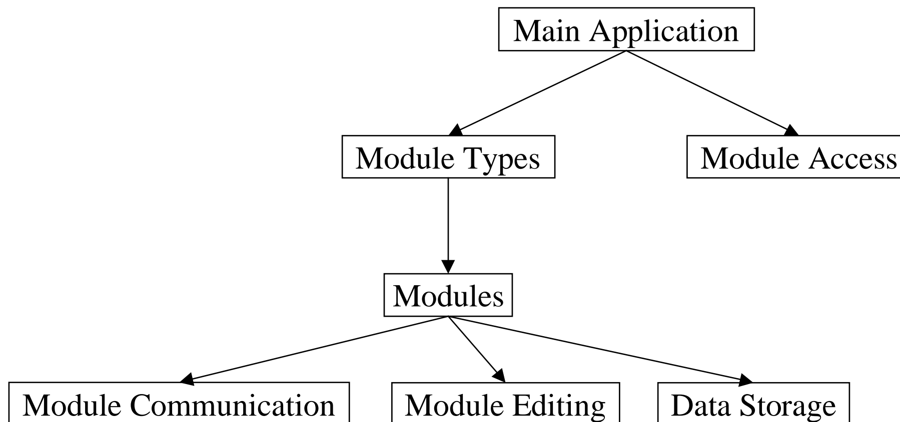

Figure 2: Basic Architecture Structure

In addition, the architecture gives a user access to the set of modules that it contains. This is so that new functional units may be added or unnecessary ones removed. General functions of modules in the architecture are also specified. Modules support some means of information-sharing or communication among them. Modules also support some form of user interaction or editing capability. Lastly, modules should permit a user, where necessary, to store settings or parameters so that he or she is not required to define them again.

By defining these module functions abstractly, we have simplified the difficult process of designing a tool to meet the prescribed goals. This architecture is a useful solution for many systems. One example of an application that is well-suited to this architecture is a VLSI design package in which different components may be used together to define ICs, and whose performance must be evaluated.

2.2. Overview of the Möbius Tool

The Möbius framework is also a good candidate for implementation within the architecture. Modules can easily be created to implement different formalism types or solvers. Further, the resulting tool would be useful for research since new modules could be created quickly and easily to test or to prove different research applications. The remainder of this chapter shows how the architecture just described was applied to the task of designing a tool that implements the Möbius framework. In implementing the Möbius tool, we will show not

only that the architecture is useful for defining complex applications, but also that it is possible to implement solvers that interact with the Möbius framework's abstract notion of model specification.

The Möbius tool's purpose is to enable model construction through the notions defined in the Möbius framework, and to allow for the solution of these models. This section introduces the Möbius tool, and shows how the tool is an implementation of the architecture by discussing how the architecture's requirements for module types, module access, and a main application are met. Further, it describes how the tool incorporates the notions of model specification defined by the Möbius framework.

2.2.1. Module access

There are many facets to the Möbius tool, but one of the most fundamental functions that it must be capable of performing, as an implementation of the architecture, is that of providing a main application for starting the modules contained within it. Each module accessed through the main application has a graphical user interface (GUI) for specifying models or solver parameters, and contains some system-specific functions for saving, undoing, and other similar functions that users expect. Part of the difficulty in providing a module-launching facility is that the tool needs to be able to launch modules without needing to know anything about the modules themselves. Further, users, whether they are researchers or system analysts, should be able to add new modules, and the tool must act appropriately on them. More specifically, the tool needs to be informed of which modules are available to a specific user, since the Möbius tool should not have to change at all in order to incorporate a different user's formalisms or solvers. Actual implementation details of how these functions are made possible within the tool are described in Chapter 3.

Once the means of module access is defined in the main application, a selection mechanism for choosing among the modules must be provided so that a user may specify which module to run. In order for the tool to be user-friendly, there should be some level of abstraction protecting the user from having to have any knowledge of class hierarchies or implementation-specific information. Consequently, upon the user's selection, the tool must determine which module the user specified and start it.

2.2.2. Module overview

Möbius also needs to provide some way of grouping modules together into module types as specified by the architecture. In the tool, these types are also the keys to defining the relationships between the models created by the modules. One of the first tasks that was required in implementing the architecture was to determine exactly what these different module types were and how they would interact. The Möbius framework has a well-defined set of formalism types that break models in the framework into predefined functional units. Consequently, many of the module types in the tool were taken directly from the framework's set of formalism types. The module types in the Möbius tool are atomic model constructors, composed model constructors, reward model constructors, study constructors, and solvers. Although studies are a trivial kind of connected model, more complicated notions are not yet supported. Due to the tool's extensibility, it should be simple in the future to include a more complicated connected model constructor. Further, the notion of solvers exists in the tool, whereas the framework has no notion of solution. This is because although the tool uses the framework's notion of modular model specification, it also is designed for use in analyzing the behavior of these models. The solver modules are used to determine performance measures over the model.

Each module type has specific features and characteristics. For module types that represent the Möbius framework's formalism types, the descriptions are very similar to the descriptions of formalism types found in Section 1.2. The following list describes each module type's important characteristics:

- **Atomic Model Constructors**: Atomic models are the basic building blocks of any larger model. Their constructors allow the specification of structural models in terms of state variables, actions, and/or other formalism-specific modeling constructs, such as input or output gates in SANs.
- **Composed Model Constructors**: Composed models are a means for creating larger models from the atomic model building blocks. Consequently, composed model constructors allow users to specify composed or atomic submodels that are contained within the constructed model. No new actions or state variables are defined in a composed model constructor. Instead, users are presented with types

of formalism-specific functions that may be used to define the composed model functionally in terms of the submodels it contains.

- **Reward Model Constructors**: Reward model constructors may vary widely, but they all must contain means for specifying measures of interest in terms of the underlying model's state. In addition, these constructors may provide a means for specifying additional state variables that are required for the correct determination of those measures. Path-based reward model constructors are an example of a case where this type of functionality may be necessary.
- **Study Constructors**: Study constructors provide a way for parameters on models to be altered quickly and easily. From a user's point of view, the model's action and state variable relationships are identical, but through changes in the model parameters, systems can be analyzed or compared under many different conditions. In the Möbius tool, these parameters are called *global variables*. These global variables may have been defined in any of the atomic, composed, or reward model constructors that were used to define the model.
- **Solvers**: Solvers are modules that execute a model such that the overall system behavior may be determined. Through a reward model, solver modules work in a variety of ways to calculate the system's specified performance characteristics. In general, all solvers take some model as input and generate some form of results. The format of these results may differ from solver to solver. Examples of solvers include simulators and state space generators.

Figure 3 shows how the different modules are expected to access each other in order to obtain the information necessary to function. This figure shows the type of module allowed to provide input to each module type. These input requirements stem naturally from the definitions of the module types shown above. In particular, atomic model constructors do not require any other models as input, while composed model constructors need atomic models or other composed models as input. Reward model constructors require that some model be defined; that model may have been specified in either a composed or an atomic model constructor. Study constructors require access to reward models. Solvers use the output of study constructors and perform whatever task is necessary to determine the system behavior

specified by the reward model over the defined submodel with the parameters set according to the study. Solvers may also use the output of other solvers. For example, many numerical solvers require a state space in order to calculate results.

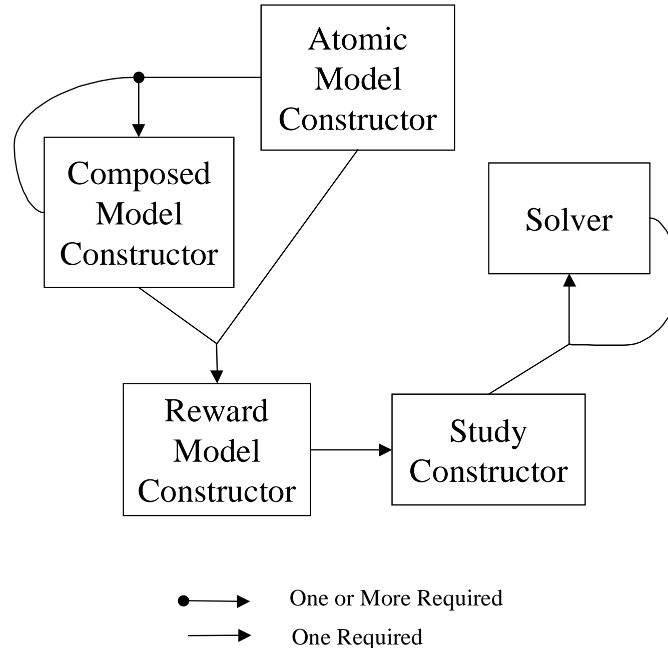

Figure 3: Module Relationship Diagram

In addition to providing an implementation in which the models may be accessed by each other, the Möbius tool also defines the language through which these models communicate. Every module within the tool is required to support certain interaction specifications so that other modules may obtain the information they need. In this way, the models constructed by these modules may have extremely different implementations, notions of state, or types of action; but as long as their corresponding modules support the tool syntax, any other model module will be able to interact with them.

2.2.3. Rapid application design

The Möbius tool is implemented in such a way that new modules may be easily created. In order to allow researchers to integrate new modules into the Möbius tool quickly and easily, much of the basic functionality has already been written in a set of base classes. By implementing new modules as subclasses of these prewritten classes, a new module writer or formalism designer has relatively little work to do to include his or her research results in

the Möbius tool. In fact, since much of the tool's functionality is independent of specific modules, these predefined base classes are able to contain all of the functionality common to all modules. These predefined classes thus allow a designer to concentrate on his or her particular module functionality, without requiring a deep understanding of the tool's inner workings. Already defined within these base classes is all of the functionality that makes the look and feel of different module GUIs similar, maintains model dependency information, and handles file menu functions such as saving and opening models. More detail on how this was done, as well as exactly what functionality is already supplied in these base classes, is contained in Chapter 4.

2.2.4. Implementation overview

The Möbius tool architecture is shown in Figure 4. As the figure shows, there are three main components to the tool. They are the control panel, the module implementation,

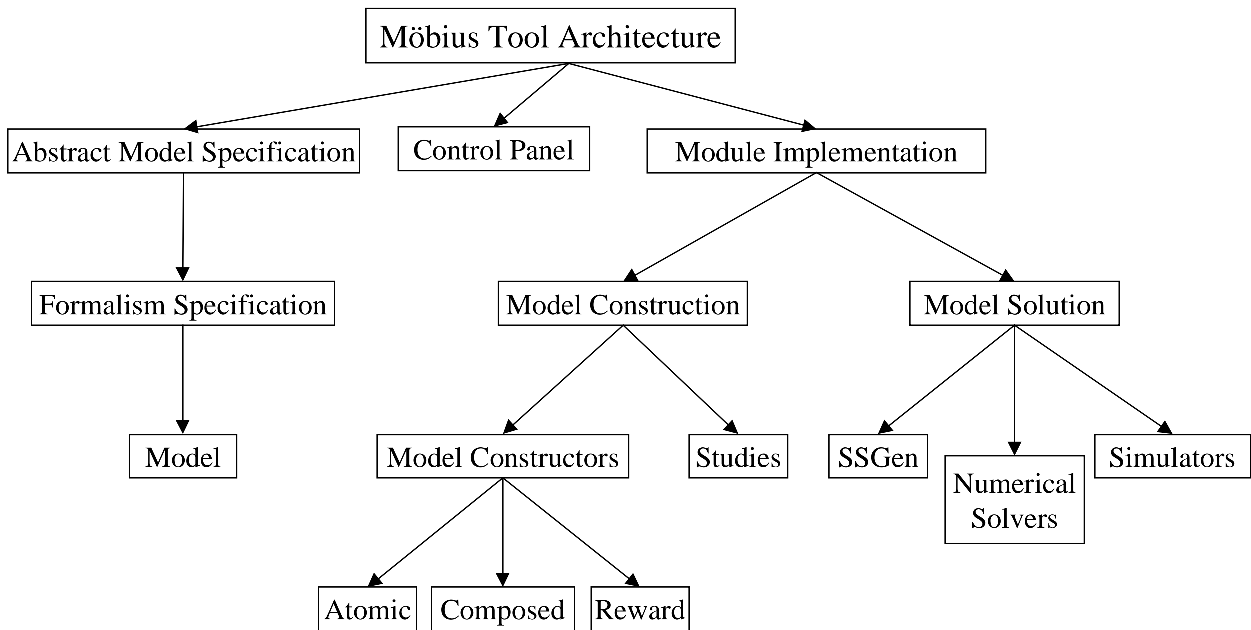

Figure 4: Möbius Tool Architecture

and the abstract model specification. In this section, these components will be described in order to give the reader a relatively complete understanding of how the tool works. The remainder of this thesis will be a more detailed description of the module and control panel

implementations. An in-depth discussion of the tool's use of the abstract model specification may be found in [8].

The control panel is the tool's main application. Within the control panel is the functionality not only for launching new modules, but also for doing so in a completely module-independent way. Further, the control panel organizes module access according to module type, and provides a means for adding new modules or upgrading existing ones as the extensible architecture requires for extensibility and maintainability.

The abstract model specification is a set of classes that implement the notions of model, state variable, and action in a formalism-independent way. By deriving all formalism specification classes from these abstract classes, formalisms may interact with one another or with solvers. This interaction is enabled by certain virtual methods that must be overloaded in order for the formalism specification to be nonabstract. Once the formalism specification has been created in terms of the abstract model specification, models may be specified in terms of the formalism.

Within the tool's architecture, the abstract model specification is defined in a set of abstract C++ base classes. Methods that are formalism-specific are defined as virtual in these base classes so that they may be completely formalism-independent. Formalism specifications are a set of formalism-specific classes that are derived from these base classes, and that overload the proper virtual methods. The abstract notions of state variable and action in the base classes may be given formalism-specific implementations in the formalism specification. For example, in queuing networks, the formalism specification for actions would be servers, and for state variables would be queues.

Module implementation in the tool is divided into two categories: model construction and model solution. Model construction is further subdivided into module types similar to the Möbius framework's formalism types, such as atomic, composed, and reward model constructors. The notion of studies is also incorporated in model construction. The tool also introduces modules that are designed to solve the models generated with the model constructors. These types of modules include state space generators, simulators, and numerical solvers.

The initial tool implementation supports a single atomic model constructor (based on SANs), a single composed model constructor (based on a replicate/join formalism similar to the one found in *UltraSAN*), and a single reward model constructor. The tool also contains two types of study constructors, for specifying global variable ranges as either sets or functions. Lastly, the tool supports a state-space generator [9], several numerical solvers, and a distributed simulator [10]. In the future, more constructors and solvers of each type will be added.

Each module type may be thought of as performing a specific transformation on a model. Figure 5 shows the functional transformation that each module type performs on a model. A structural model, as specified by a user, is turned into an executable model, which is its software equivalent, by each of the model-constructing modules. Each of the solvers then takes a model and generates results. For example, a simulator or a state-space generator uses an executable model to generate results. The state-space generator's results are a behavioral model that numerical solvers may use to obtain results.

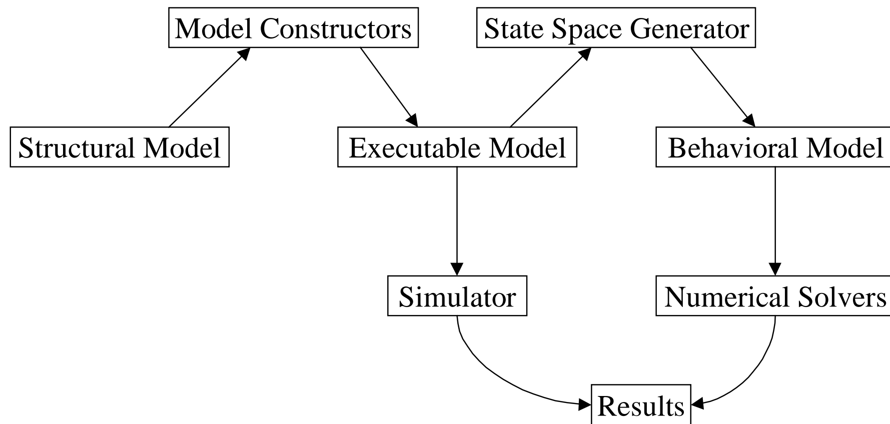

Figure 5: Model Transformation Process

This transformation process is performed through compilation and linking of C++ class files that are a particular executable model. Figure 6 shows this process. Specifically,

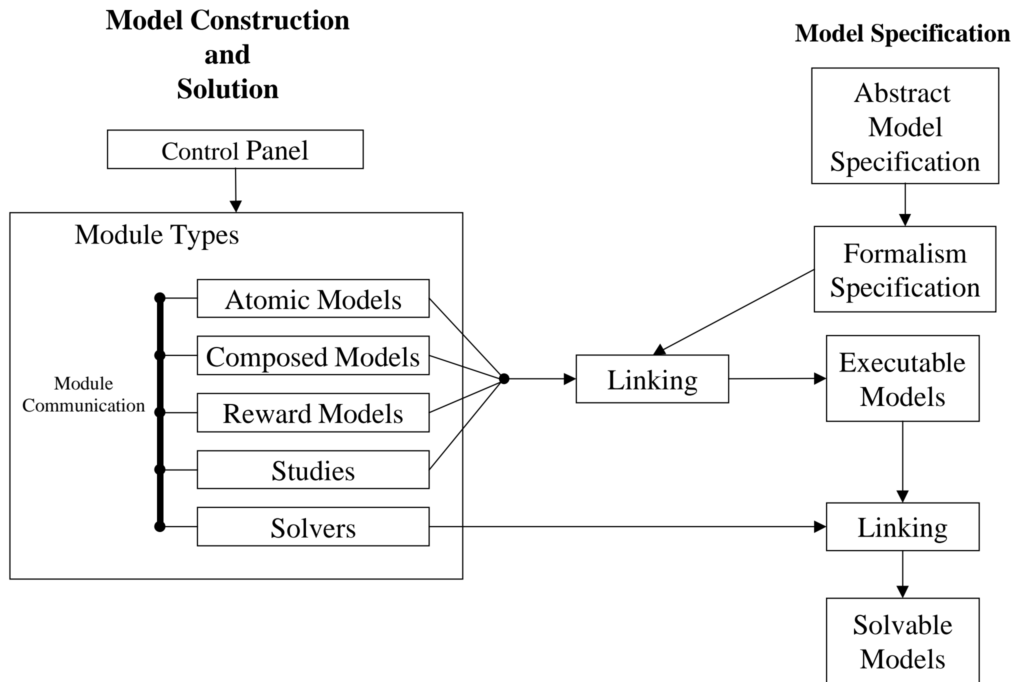

Figure 6: Model Transformation Implementation

when constructing models, information is passed between modules through predefined methods and data structures that must be supported in order for any module to be usable. In addition, an implementation of the formalism itself is created in terms of the abstract model specification. The structural model may then be transformed through the model constructors into the terminology of the formalism specification.

To perform the transformation, model constructors, which are implemented as a set of Java classes, support methods that generate model-specific classes derived from the formalism specifications. When the formalism specifications are linked and compiled with the model specifications, an executable model is generated that is identical to the structural model the user input, but cast in the correct formalism.

Solvers take the executable model specifications of the model, and by linking together object files for reward model specification, atomic models, compositions, and studies, generate a solvable model. Solvers may then interact with this solvable model to determine the specific measures of interest. This solver/model interaction is done through methods specified in the abstract model specification such that any solvable model that a solver is

presented with will support them. For example, the interface for SANs requires that a user specify SAN-specific information such as initial place markings and activity rates. All of this information is then used to create a textual output class that is derived from a C++ version of these SANs. Then the generated code is linked and compiled with the library's SAN formalism, and an executable self-contained model is created. By "self-contained," it is implied that all of the execution and behavioral knowledge concerning a specific model component is contained within this executable. A similar process occurs for other atomic models, composed models, reward models, studies, and solvers.

Note that many C++ executable models may be linked into numerous solvable models, just as the structural model defined in a module constructor may be used in numerous larger models. This is how the specification and solution of models is accomplished, and how both parts of the tool ensure the correct interaction and the modularity of models while implementing the Möbius framework's abstract notion of model specification.

3. THE CONTROL PANEL

3.1. Introduction

The control panel is the Möbius tool's main application. It organizes the accessible module types, as well as the model construction and solution modules of each type. It enables a user to run multiple modules at the same time, and provides users with a simple means for opening or creating models through these modules. Further, it implements caching strategies to make model access fast.

The control panel is a window that consists of a menu bar and a status text bar. The menu bar is used for performing functions like creating, opening, and closing models and exiting the system, and the status text is used to display current system functions or status. Figure 7 shows a picture of the Möbius control panel running under Windows NT.

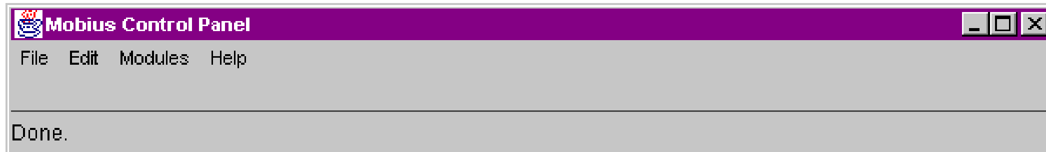

Figure 7: The Möbius Control Panel

Because the control panel is implemented in Java, its appearance may differ on other platforms, but its functionality on all platforms is identical. This chapter is devoted to explaining how the control panel provides module access and categorization of modules, and the important features that make the Möbius tool useful for the construction and solution of models.

In this chapter, there will be numerous references to Java-specific terminology and characteristics. For completeness, some important terms will be defined here, but for a more complete understanding of Java, see [11]. First, a *class* refers to a set of data elements and a group of methods that act on them. Furthermore, *packages* in Java are a collection of classes organized together because of their similar functionality. For example, the `java.awt` package contains basic visual components that may be used to define interfaces, and the

`java.lang` package contains Java's most central classes. *Components* are classes that implement graphical user interface objects. List boxes, windows, and menu items are all examples of components. In addition, Java is *interpreted*, which means that Java is compiled for a Java Virtual Machine rather than for a specific architecture. This allows Java to be platform-neutral, among other things. Further, Java is a dynamic language, in the sense that Java classes may be loaded into and instantiated in a running Java interpreter at any time. Lastly, *serialization* refers to the process of writing an object's complete state to an output stream. This enables a currently running class to be represented by a file. *Deserialization* refers to the opposite process of creating and loading a class from an input stream.

3.2. Initialization

The first time the Möbius tool is started by a particular user, the `.mobius` directory is created by the control panel in that user's home directory. Details of the content of this directory are found in the appendix. In short, this directory contains a `modules.cp` file that specifies the menu set up for that user and the class files corresponding to different modules, which are associated with specific menu items. If the Möbius tool is being run for the first time, and the `.mobius` directory was just created, then the control panel will have no accessible modules. The modules that a specific user requires access to may be added to the control panel, however. When the control panel is closed, or, equivalently, the tool is exited, the `modules.cp` file is written such that the most recent accessible module configuration is always saved.

The `modules.cp` file contains lines of the format:

```
<Module Type>;<Class Package>.< Class Name>
```

where "module type" corresponds to the specific module's category and "class package" refers to the Java package containing the class file specified as "class name." The actual class that is identified here is to be an implementation of the module launched by the control panel. A more detailed discussion of the means by which these modules are associated with menu items and launched in the control panel is given in Section 3.3. The following is an example of the format of each line found in the `modules.cp` file:

```
Atomic Model;Mobius.san.SanInterfaceClass
```

Another initialization file in the `.mobius` directory is the `upgrades.cp` file, which specifies version upgrades so that the control panel may automatically recognize when code updates or module upgrades have been performed. Details on how this file is used are also contained in the following section.

3.3. Module and Model Access

This section describes the ways that both modules and specific models are accessed in the Möbius tool through the control panel. As stated in the previous section, it is possible for users to change the modules that are accessible to them. Models are module data in the Möbius tool, and they may be in one of three possible states. They may exist on disk, they may be opened for editing within a module, or they may be stored within a loaded module, which is not being edited. *Loaded* modules permit the quick communication of model data whereas *opened* modules permit the editing of model data.

3.3.1. Creating and opening models

Modules are accessed through the file menu on the control panel. It is possible to either create a new model or open an existing one through each type of module. Since models are module data, the saving and opening of models may be performed in a module-independent way through the serialization of the entire module. This is flexible, but it may write a great deal of extraneous information in addition to the model unless the module serialization methods are overloaded such that only the essential data within the module is written. Since Java already defines the `writeObject()` method for this purpose on all serializable objects, there was no need to implement special methods within the Möbius tool to facilitate the overloading of object serialization methods.

Access to specific module types for model construction is done through the control panel's file menu. Figure 8 shows an example of the submenus that may be encountered by the user while trying to create a new model. This menu structure organizes modules according to the formalisms they represent, and each formalism or module is associated with the correct module type that categorizes it. The submenus encountered while an existing

module is being opened are identical to those found when trying to define a new model, but the function performed upon the final menu item selection is different.

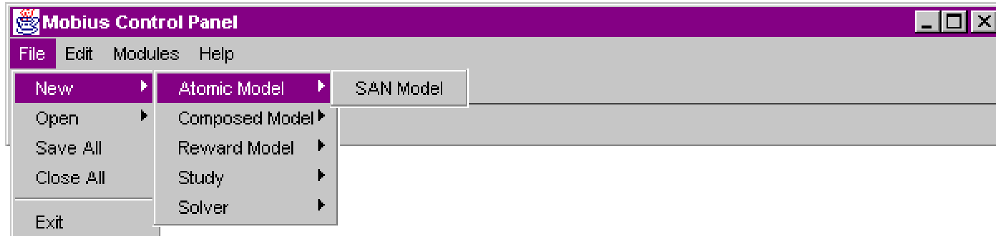

Figure 8: Control Panel Model Access

As the figure shows, the first submenu encountered consists of a list of all the module types. Upon selecting from this menu, the user is presented with all of the formalisms or accessible modules of that module type. While trying to create a new model, the user's final click on the module menu item will launch a new version of the correct module. If the user wants to open an already defined model, Möbius will display a file dialog through which the user may select the saved model to load into the correct module.

This process is not as straightforward as it might seem. Recall that the Möbius tool needs to be easily extensible. Consequently, it is not possible to simply hardcode the correct formalisms, or modules, in the control panel. Instead, the control panel needs to operate on modules in a generic way. This is a case where Java's interpretation and dynamic class loading become useful. Several data structures enable this functionality and flexibility.

Within the control panel there are two important lists. First, the `Types` list is a list of the allowable module types (atomic, composed, etc.). The other, the `Classes` list, is a list of lists of specific modules. Figure 9 shows an example organization of these lists.

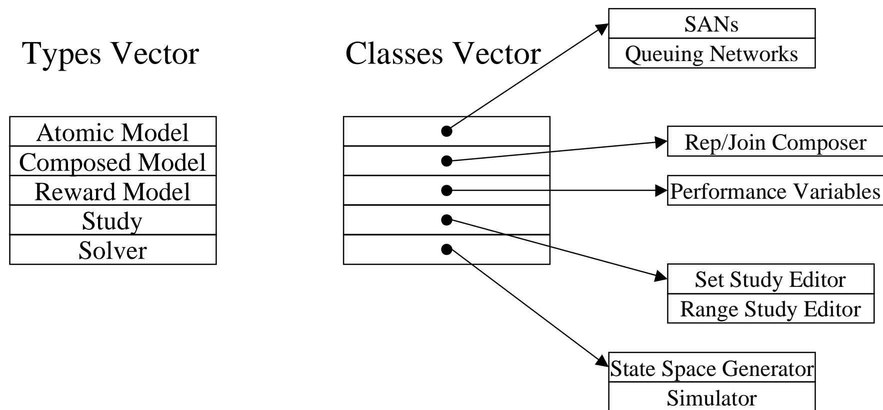

Figure 9: Type and Class Lists

The organization of the `Classes` list is identical to that of the `Types` list, but each sublist contains formalism-specific modules. Notice how these data structures mirror the file menu new and open submenu hierarchies. In fact, the new and open submenus are created from these lists. Thus, when the user makes a selection, it is simple to find the corresponding formalism module in the classes list.

When the user wants to create a new model of a particular formalism, the *Class* object for that formalism is accessed. This `Class` object is a special object, one of which exists for each class loaded into a Java Virtual Machine. Normally, to create a new class, we would define an object of the correct type and set it to reference the proper object by calling `new` with the correct constructor. Using these `Class` objects, we do not need to know the constructor, and consequently, the control panel never needs to know the class name of the module the user wants to launch. That is the mechanism for creating new models in a generic way. Any module's implementation class has a corresponding `Class` object. This `Class` object is accessed, and its `newInstance()` method is called. This method returns a new instance of the correct class, which was created using the class's empty constructor. Then some processing, which is described below, is performed, after which the module's editor is shown, and the user is free to edit the new model of the specified type.

When the user wants to open an existing model, the corresponding module class is used, but its `Class` object is ignored. Instead, an open file dialog is presented to the user. Each formalism module class has a unique extension by which appropriate model data may be

recognized, and the file filter is set to files of that type. After a file is selected, the file name is passed to the formalism method `interfaceNameIsValid()`, which is able to determine whether the specified model file is truly a serialized class of that formalism type. If the model is acceptable, then it is deserialized, some processing is performed by the control panel, and the module's graphical user interface is shown for the user to edit.

Two things are done during the control panel's processing stage prior to displaying a module's editor. First, the module's reference to the control panel that launched it must be set. In a new model, the reason the reference needs to be set may be obvious, since the new model's module will have a null pointer to the control panel. In a deserialized model, it is also necessary because any old control panel reference that was written to disk would be useless to the model in trying to access the new control panel that opened it. In fact, the `ControlPanel` variable for a module's interface is declared `transient` for this reason. *Transient* is a Java keyword referring to a field that is not serialized with the rest of the class. There is no reason to write the information, since it will be meaningless upon deserialization. The second thing done prior to displaying a module's editor is that the control panel caches the interface. This is discussed below. After the control panel has performed these two steps, it presents the module's GUI to the user.

The deserialization of model data requires noticeable time, since the entire file must be read and a corresponding object instantiated. It would be undesirable for a user to have to wait for this deserialization to take place every time he or she attempted to load a model for interaction. Therefore, the control panel implements caching strategies so that recently accessed models are only deserialized once.

The implementation of these strategies consists of two vectors: an open interface list, and a loaded interface list. The open interface list contains pointers to each module whose editor is open. The loaded interface list contains object pointers to all modules that are not open, but have been used for information by other models or solvers. Any open interface that has an associated file is moved from the open cache to the loaded cache when it is closed. This is because once a module has been opened, it is more likely to be used by another module. These caches may each contain references to up to ten modules, and the replacement strategy simply removes the first element in the list when this bound is exceeded.

Another important feature that the tool supports in order to make the tool more maintainable is automatic upgradability. Since formalisms may be extended, or their implementations may require correction due to programming errors, the Möbius tool provides a means for upgrading existing modules. Since models are saved and opened using serialization and deserialization respectively, the original classes implementing older modules cannot be changed. If they were changed, then Java would simply throw an exception error while trying to deserialize a model in an older module version, since it would not recognize the old class format. Consequently, all upgrades require new module classes.

Once these new module classes are present, however, the Möbius tool's upgrade functionality takes place transparently to a user. This is done in two ways in Möbius: version upgrades and model patches. A version upgrade is required when there is a large change to the tool's existing classes. This may be a result of several formalism upgrades, or may be due to a change in the module base classes or supporting component packages since classes from these packages are used in all the formalism modules. Model patches are local to a single formalism and will usually be due to bug fixes or formalism extensions.

The mechanism for version upgrades requires several changes. First, it requires a new main class or a replacement to the old one if the control panel class has been changed. This *main class* is the class that contains the `main` method. Only one class may contain this method. This class's only function is to start the correct control panel class with the existing `Runtime` object. If the base classes or utility classes have been changed, new versions of all formalisms should be generated in new version packages. In this way, all of the old formalism classes will remain valid.

Once the control panel has been instantiated, it will look for an `upgrades.cp` file in the user's `.mobius` directory. This file has lines of the form:

```
<new class package and name>;<old class package and name>
```

where each line represents an outdated class file and the new class file that replaces it. The control panel uses this information to alter the menus such that all old formalism classes in the `Classes` vector are replaced with the corresponding upgraded formalisms.

For model patches, the upgrades file can be used, or the upgraded module may simply be added to the control panel like any other module. A list in each module, the

`upgradeList`, is the list of other module classes for which the module is an upgrade. When a new module is added, any formalism for which it is an upgrade is first removed.

When any model is opened, all of the modules in the `Classes` vector are searched to see if any of them are upgrades for the module that originally constructed the model. If an upgrade is found an automatic model data conversion process is begun. This process is described in detail in Subsection 4.2.7.

Upgrade functionality is included in the Möbius tool to satisfy the need for extensibility and flexibility. Formalism upgrades and tool enhancements should not invalidate existing model data. Both the version upgrade and the single formalism upgrade are designed to avoid any such problem, and make the inclusion of new research and the correction of existing errors as simple a task as possible.

3.3.2. Adding and removing modules

In order for the Möbius tool to be extensible, the module types were implemented in a dynamic way. Different users may have access to completely different module types, and consequently, different modeling formalisms. The control panel allows users to both add and remove modules to or from any module type on the file menu.

The control panel's "Modules" menu provides access to two interfaces, one for adding new models and one for removing existing modules. In order for a new module to be added, an "add modules" dialog window is used, which requires the user to enter the desired module type of the new formalism. This dialog window allows the specification of which submenu will contain the launching menu item for the formalism. It is a user's responsibility to ensure that the formalism also meet the definition of the module type to which the user is adding it. The user must also specify the package containing the module, and the name of the class file itself. This is enough information for Java to load the class dynamically from the file and properly place it in the `Classes` vector. Consequently, no modifications to the control panel are necessary for it to handle existing or future formalisms as long as they are of one of the predefined module types. Similarly, the functionality to remove modules provides an interface through which module classes may be taken out of the `Classes` vector.

3.4. Supporting Classes

In addition to the classes that specifically implement modules and the control panel, many additional classes were implemented in order to build them. These simpler classes or supporting classes are not restricted to use in the Möbius tool, although they were developed for it.

3.4.1. Utility classes

The utility classes are the most basic classes implemented for the tool. They are the tool's basic support for modules. There were two main goals in the development of these classes: a consistent look and feel across modules, and a way to correct Java's library class inadequacies. These classes may be found in the `Mobius.Utils` package.

First, a consistent look and feel is useful in making a user comfortable with an application. Further, it reduces the time it takes a user to learn how to use future modules if he or she is already familiar with the types of interface components he or she will encounter. These classes include lists, choice boxes, text fields, buttons and other visual components. Since all modules are expected to use these classes, it is easy to define default sizes, colors, fonts, and other properties for components. This allows a consistent look and feel to all of these components on any interface. It also makes the interface developer's job much simpler, since these classes are already provided. Further, if Möbius's look and feel is to be changed, then these classes may be altered such that the changes will automatically be propagated to any existing modules.

Second, although Java offers many useful features, such as platform independence, interpretation, and object orientation, it is still a very new language. As such, it is far from bug-free. During the development of Möbius, numerous problems were discovered in the `java.awt` package. Platform independence has still not been fully realized, and in the most severe cases, exceptions actually occur in some classes. For example, attempting to remove all of the elements from lists or choice boxes that were empty in the JDK for HP's caused exceptions. Through the definition of the utility classes, these errors were corrected so that they do not appear in any Möbius modules.

In addition, the utility classes have been designed to look correct on the multiple platforms for which Möbius is supported. This was necessary because fonts appear different on different platforms, and because the components themselves have slightly different appearances from platform to platform. In some cases, the differences are great enough to cause interfaces to be unusable. Some components may be pushed off the visible area of the windows that contain them. Through the utility classes, different fonts and default component sizes enable interfaces to appear correctly on Windows NT, Linux, Solaris, and HP-UX systems.

In other cases, some classes were used in many modules, but did not exist anywhere within Java's standard libraries. For example, in Java, a *vector* is a dynamically sized list of objects, but basic types are not objects. Consequently, it is impossible to make a vector of `ints`, `booleans`, `floats`, or any other basic type. Further, since Java has no pointer types and removes address access to variables, the use of objects is the only way to pass parameters by reference to methods. For both of these reasons, Möbius classes representing basic types are defined such that they are derived from the `java.lang.object` class.

3.4.2. Dialog windows

Dialog windows are also implemented as supporting classes for the tool, and may take many different forms. Examples of dialog windows include OK/Cancel windows and Open File dialog windows, among many others. These types of window are extremely useful for quickly obtaining responses from users or for displaying important information. The `Mobius.Dialogs` package contains a useful base dialog from which others may be rapidly derived, as well as numerous simple dialogs that most modules require. These include Yes/No, OK/Cancel dialogs and a list dialog for displaying large amounts of information to the user. These classes are provided both to standardize the look and feel of modules by presenting users with a consistent set of dialog windows, and to enable application designers to implement new dialogs with minimum effort.

One of the properties of dialog windows that make them useful is described by the term "modal." If a dialog is *modal*, its process is blocked until the window is disposed. For a user, this means that while the dialog is being displayed, other windows in the same process

are inaccessible. Since the Möbius tool runs in a single process, any modal window may be used as a way to get a response from the user by preventing him or her from doing anything else within the system until the response is provided. This ability of dialog windows is quite powerful, and is put to use in many parts of the tool.

4. CLASSES TO SUPPORT MODULE IMPLEMENTATION

This chapter presents the implementation of the classes that define the modules in the Möbius tool. The abstract module classes for the tool were designed to support the formalism-independent functions that the Möbius tool provides. Modules are thus required to overload formalism-specific methods in order to implement the desired formalism functionality for model construction or solution as necessary. Further, these base module classes support many functions that make the development of specific formalism modules or solvers easier.

4.1. Module Organization

Modules are the functional units in the Möbius architecture, and the means for model construction and solution in the tool. Modules in the Möbius tool are composed of three main classes from the `Mobius.BaseClasses` package. A module, as shown in Figure 10, consists of an interface, an editor, and an information class. In Sections 4.2, 4.3, and 4.4, each of these classes is described in detail, but a general overview of their relationships will be given here.

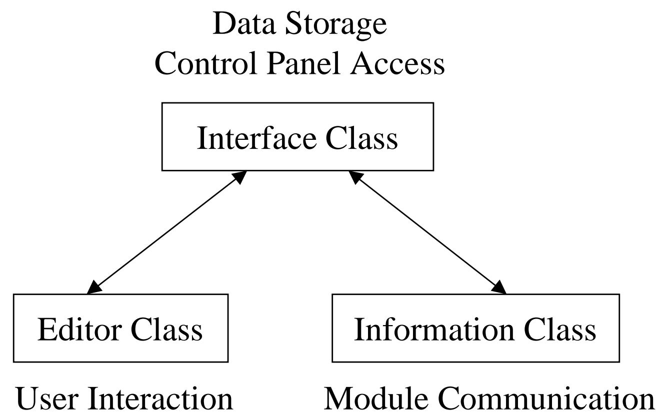

Figure 10: Module Composition

A module may be thought of as a single entity, since none of the three classes that define a module will ever be found without the other two. There are three main module requirements defined by the Möbius architecture, and each of these module classes

implements one of them. The interface class is the main class through which model types are recognized and the module itself is referenced. The editor class is the main GUI for a module, and provides module editing. The information class is the part of the module that specifies the means for communication with other modules. By organizing module functionality through separate classes, the process of developing new module objects is simplified.

The interface class contains references to both the editor and the information classes, each of which in turn has references to the interface class. The editor and information classes are expected to use the interface in order to reference each other. When a programmer wants to create a new module for use in the Möbius tool, the first step is to create a class derived from each of these base module classes. Each of these classes has some methods that may and some methods that must be redefined by the derived class for the module to be non-abstract.

The interface class is the main component of a module. It is through the interface class that a module specifies its function in the tool. In the case of model constructors, this is a specification of which formalisms constructed models belong to (for example, whether a module constructs SANs or replicate/join composed models), and in the case of solvers, it is a specification of the type of solution method the module performs. Further, it is through these interface classes that model dependencies are maintained, model compilations are performed, upgrades are enabled, and the means of model storage is specified. The mechanisms by which interfaces are implemented and their interactions are performed are described in Section 4.2.

The editor class is the means of user interaction with a model. Each interface needs a corresponding editor in order to allow a user to define the model, solver parameters, or other types of variables associated with that type of module. Many components have already been provided in the supporting classes to make designing an editor simple, and to ensure that all editors have a consistent look and act similarly. More detail on editor classes and the components provided to ease their development are in Section 4.3.

The information class is the class through which modules communicate. Many methods must be defined to ensure that a derived information class is nonabstract. The types of interaction performed through the information class may include determining model information such as the names and types of state variables in a model, or the names of global variables defined in a model. System-specific information, such as the name and path of the

makefile to use when compiling a structural model into an executable model, is also communicated through the information class. A deeper discussion of the functions required in an information class, and how modules interact through them, is provided in Section 4.4.

4.2. Interface Classes

The interface class is discussed in more detail in this section. It is through the interface class that different formalisms are recognized and different models are accessed. The base class for these interface classes in the tool is `Mobius.BaseClasses.BaseInterfaceClass`. In order to simplify the design of new formalisms and modules in the tool, the base interface class implements several important functions that are used by the system transparently to all derived interfaces. This makes any new formalism, study editor, or solver work correctly within the Möbius tool with very little additional effort on the programmer's part.

4.2.1. Dependency maintenance

One of the main functions of the interface classes is to keep track of model dependencies. Recall that one of the goals of the Möbius tool is to allow for modular design of models enabled by submodel reuse. For this to be possible, some means for ensuring the total model's consistency must be present.

Before discussing how dependencies are handled, several terms must be defined. Figure 11 shows an example of a possible "dependency graph" within the Möbius tool; it will be used to help describe the important terms. A *dependency graph* is a directed graph used to describe dependency relationships between model constructors in the Möbius tool. In Figure 11, each box corresponds to a different model-specific interface class. Each interface in the graph is referred to as a *node*. A *parent node* is a node that contains the current node. A *child node* is a node contained by the current node. In the figure, the Reward1 reward model node has the Study1 and Study2 nodes as parents, and the Composed1 composed model node as its only child node. A *root node* is a node with no parents. In the figure, the Study1, Study2, Study3, and Reward3 nodes are all root nodes. A *dependency tree* is a set of connected nodes in the graph terminating at a root node.

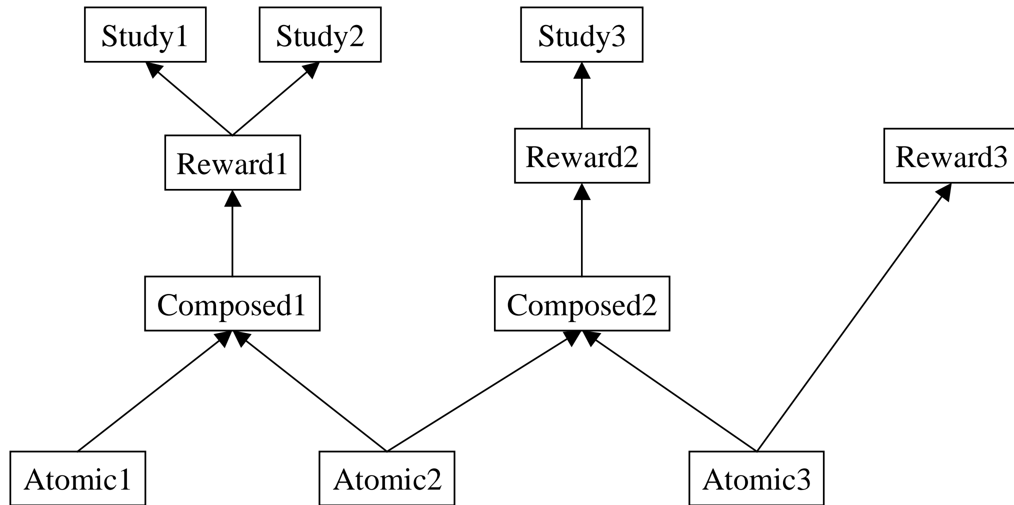

Figure 11: Dependency Graph Example

Figure 11 also shows some important characteristics of the way the Möbius tool provides model modularity. Any given model may be contained in multiple dependency trees. For example, the atomic model interface, Atomic2, is contained in three distinct dependency trees, each of which has a different study node as a root. Further, although solvers require studies in order to determine the model's behavior, any type of model-constructor interface may be a root interface in a dependency tree. In this figure, both studies and reward models are shown as possible roots.

At each interface in the tree, two important pieces of information are stored in order to maintain dependency information. Specifically, each node stores a copy of the root nodes for each tree that it is in, and the subtree of which it is the root. For example, Figure 12 shows the dependency node information that is stored for the Composed1 node from Figure 11. This information is sufficient for the determination of any dependency information important for maintaining model consistency.

Each dependency tree represents a particular model. In order for a model to be solvable, the Möbius tool requires that the model have a single study for global variable specification, and a single reward model for performance measure specification. The reward model requirement is included so that the model will have a reward structure, and the requirement of a study is chosen to ensure that all parameters have been defined. Note that the tool does allow composed and atomic models to be used in multiple larger models

multiple times. In the future, the tool may be extended to allow more complicated model hierarchies involving multiple reward models or studies; even submodels containing solvers could be allowed for fixed point solution.

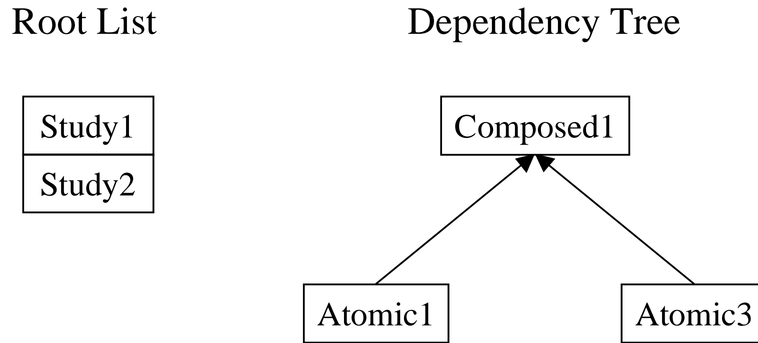

Figure 12: Dependency Node Information Example

The abstract class that implements the interface portion of modules in the tool is the `BaseInterfaceClass`. The `BaseInterfaceClass` has two specific methods that developers of derived formalism interfaces are expected to use to interact with the dependency lists. These methods are `include()` and `uninclude()`, and they each take another interface as a parameter. The `include()` method should be called when the current model contains another model. The included node then becomes a child of the node that included it. `uninclude()` should be called when the current model no longer contains any occurrences of another submodel. The `uninclude()` method removes that interface from the calling interface's child list. Because the new formalism developers are limited to using these methods, there is no direct interaction with any node's parent list. This limitation reflects the notion that submodels should be independent and unaware of the larger models that may contain them.

4.2.2. Saving

The interface class is also responsible for performing all of the correct functionality associated with saving a model. Figure 13 is a flow chart of the interface class save

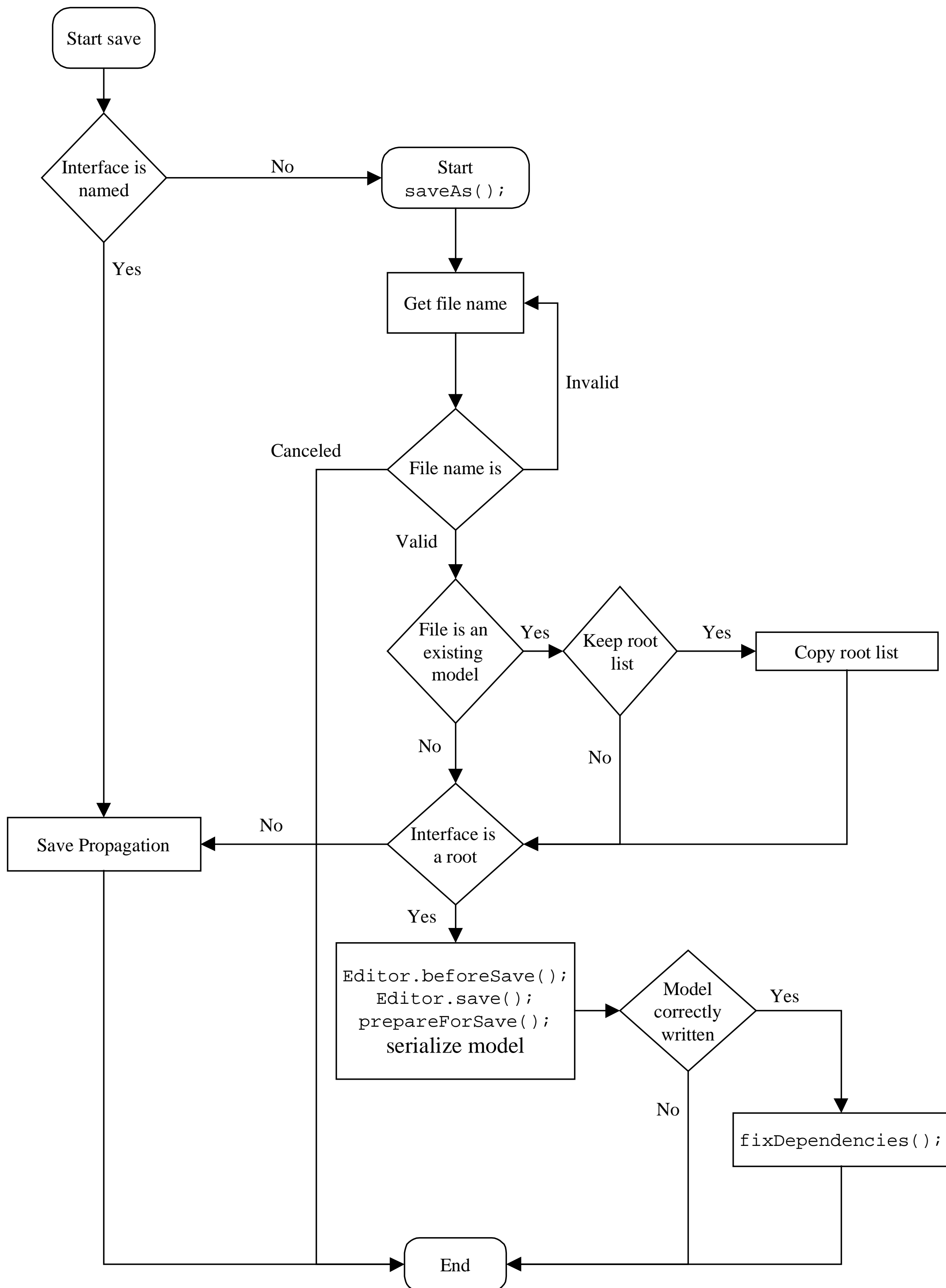

Figure 13: Interface Save Functionality Flow Chart Diagram

functionality, and is included as a reference for the discussion on the interface save implementation.

When a new model is being edited and has yet to be saved, it does not have an associated filename by which the control panel may recognize it. For this reason, each interface has an `untitledName` field to which the control panel concatenates a unique number when a new interface is created for editing. Until the model is saved this is the unique identifier for the corresponding interface.

The first time a model is saved, the `saveAs()` method is invoked. This method obtains a new file name for the model. This file name must be a valid name for the formalism type. Specifically, each interface has a particular extension specified for models of that type. The user will continue to be prompted for file names until a valid one is specified or the save is canceled. Then, the interface checks to see whether the specified file name already exists. If it does, and the user is trying to replace an old model, the old model is loaded through the control panel, and the user has the option of retaining the saved class's root list. In this way, users may replace old submodels with updated ones, and retain the proper structure of the larger model. If the old model is no longer a valid interface class, or if it was a root, then the user is not prompted.

Then, if the interface is not a root node in the dependency tree, the save is propagated up the tree. Save propagation will be described in detail in Section 4.2.4. This section focuses on the save functionality that occurs on a root interface. In particular, if the saved model is a root, several functions must be performed before the model data may actually be written to disk. First, methods are called on the GUI itself. Since this GUI will be derived from the base editor class, the interface class knows the methods will be present. These methods are `beforeSave()` and `save()`, and will be described in Section 4.3, which specifically deals with the editor class. For this discussion, it is sufficient to know that some processing is performed in the editor prior to saving, and that these methods are called. After these methods are called, the interface's `prepareForSave()` method is invoked. The `prepareForSave()` method is used to correct the child list of the current interface. In particular, the information specified by calls of `include()` and `uninclude()`, which is

kept in temporary lists prior to saving, is used to update the interface regarding the child interfaces to add or remove from its dependency tree.

After the included and unincluded interface information is correctly processed, the model is then written to disk through Java's object serialization methods. This process is considered successful if the file of the specified name exists, and the user has write access to it. If the model was successfully saved then the `fixDependencies()` method is called.

The `fixDependencies()` method is used to perform two functions. First, the children interfaces and the root interfaces are processed to see whether the corresponding files still exist, and the interfaces in the root list are checked to ensure that they are still roots. Second, the models in the dependency tree are updated to contain the current node as a root if it is one, or to remove it from their root lists if it is no longer one. After this, the save is complete, the saved interface's dependency tree and root list should be correct, and all other nodes in the interface's subtree of the dependency tree should also be correctly updated.

After a save, the model is then compiled if it is "compilable." Whether a model is compilable is determined by a simple flag that should be set if the model has been defined incorrectly. The functionality of model compilation itself will be described in Section 4.2.6.

4.2.3. Validating models

Since submodels may be used in numerous larger models, the Möbius tool needs a means for ensuring that changes in these submodels do not cause the models that contain them to become "invalid." *Invalid*, as used here, denotes a situation in which some submodel has been changed, and those changes have made some specification in a containing model incorrect. For example, if a replicate/join composed model contains two submodels that share a piece of state, and the state variable contained in one of the submodels has been deleted, the composed model will become invalid. In that case, the composed model must be changed. Different formalisms require different validation techniques. Consequently, the base interface class contains a `validateInterface()` method that returns `true` if the interface is valid. By default this method does nothing and returns `true`. Formalism module designers are expected to overload this method to correctly account for possible changes in submodels.

This method should not simply detect errors and exit, however. It should detect errors, determine the means for correcting the errors if possible, and prompt the user to accept the necessary changes if they are substantial. If the changes in a validation are accepted the parent model should be completely valid based on the definitions of the submodels it contains. Further, if changes may be validated without affecting any of the user's model specifications, there is no need to prompt the user, and validation may be performed without any user interaction whatsoever. By propagating these validations up a dependency tree, an entire larger model including composed models, reward models, and studies can be made valid automatically in response to changes in a contained submodel.

4.2.4. Propagation of interface saves

In Section 4.2.2, the save functionality was described, but the propagation of interface saves was not discussed. Propagation of interface saves occurs when a model that is not a root interface is saved. Since submodels may be included in many larger models, this process ensures that all of the containing models remain consistent despite changes in any contained models.

The first step for propagation is to determine the ordered list of interfaces to which the changes need to be propagated. The order is extremely important, and not as straightforward as it might at first appear. Submodels may be contained in larger models multiple times in different ways, and any submodel must be processed prior to the processing of any of its parent nodes. The appropriate save order is determined using the method `getPropagationSaveList()`, which returns the list of interfaces to which changes need to be propagated, in reverse order. Figure 14 shows an example of two complex models with dependency roots A and J, and a correct propagation list for changes in submodel X. Although not unique, the sequence of interface nodes X, D, K, C, B, J, A is a valid propagation order for making both large models consistent with any changes in interface node X.

It is easy to see that the propagation list shown will ensure that all models are consistent, but the algorithm to obtain the propagation list may not be obvious. To illustrate

how the propagation list is generated, we will go step-by-step through the algorithm implemented by the `getPropagationSaveList()` method.

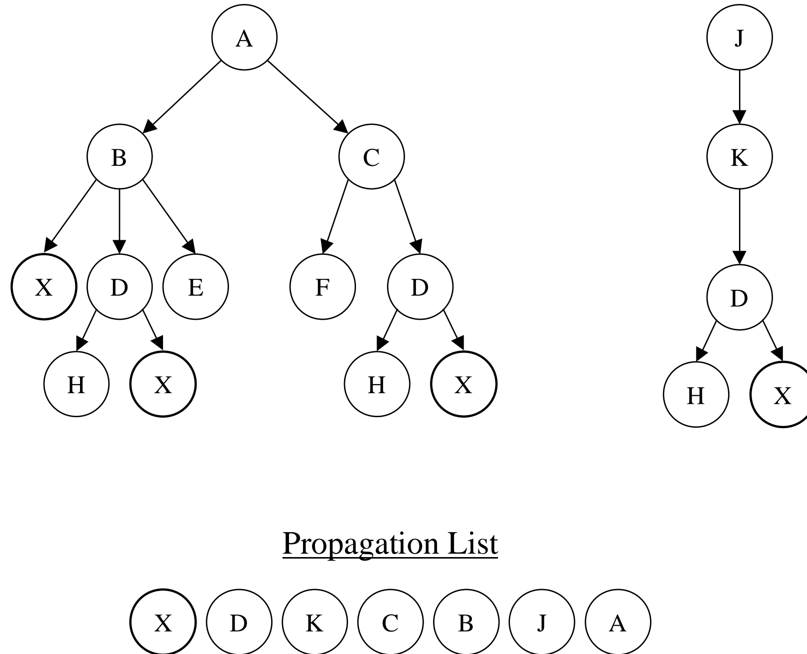

Figure 14: Example of a Propagation List

The algorithm creates an initial propagation list containing the list of root interfaces for the node initiating the propagation. This list is then processed in order for each node. At each node, any children that contain the changed model in their dependency subtree are appended to the end of the list. If a node to be appended already appears in the list, the first occurrence is removed prior to the appending of the node to the end of the list. In this way, lower-level models in the tree are processed prior to any parent models. This process is repeated until the entire list has been processed. The result is always a valid propagation list in reverse order.

For the example model dependencies shown in Figure 14, Figure 15 shows the step-by-step formation of the propagation list. At step one, the list consists simply of node X's root list. At step two, node A's children, B and C, are added to the list, since both contain model

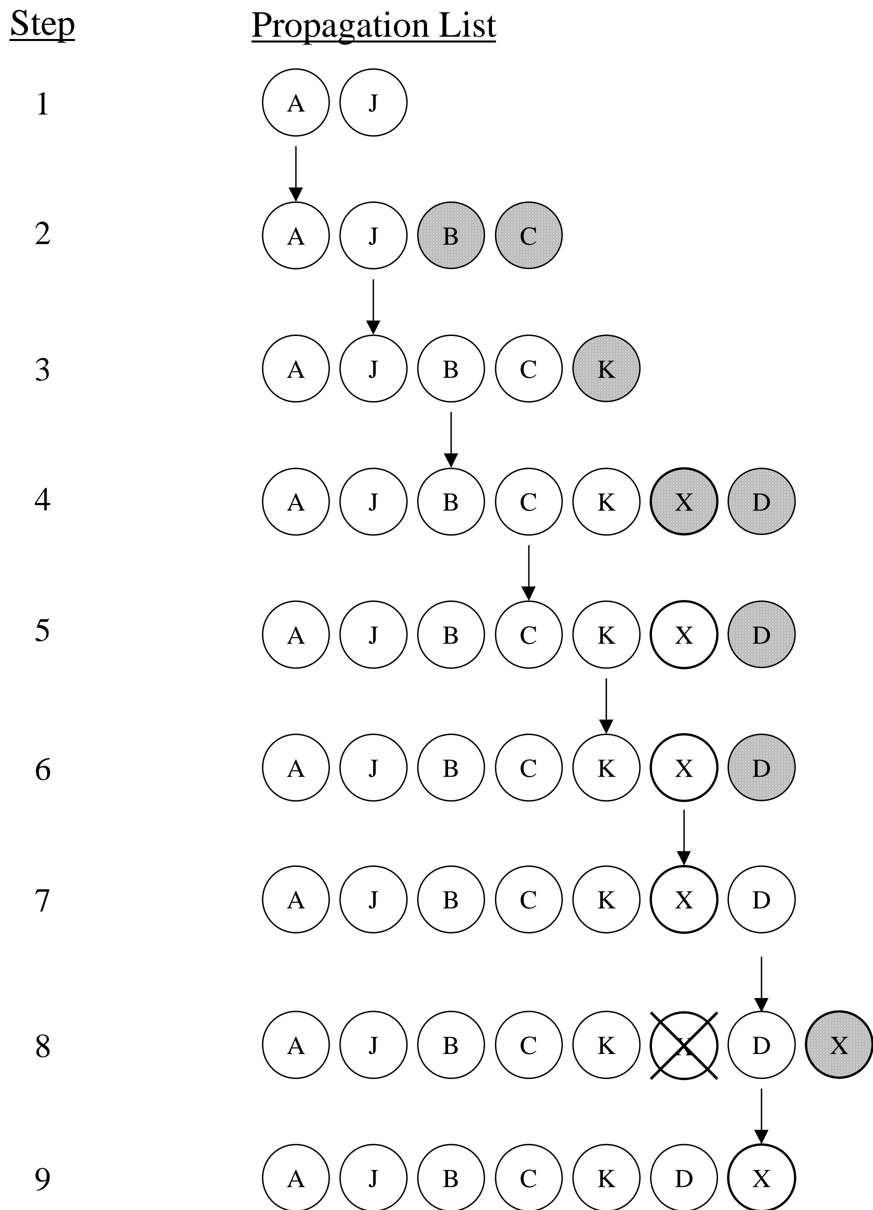

Figure 15: Propagation List Formation

X. At step three, node J's child K is added to the list, since it too contains model X. At step four, since models are considered to contain themselves, both of B's children, X and D, are added to the list. In step five, node D is removed from the list, since it was already present, and then appended. Step six is identical to step five, except that in this step, node K is processed. In step seven, the list is not changed, since node X has no children. In step eight,

X is removed from the lists and appended to the end, since D is one of its parent nodes. Lastly, in step nine, the list is again not changed since X has no children, and the algorithm terminates, because the end of the list has been reached. The resulting list is identical to the example shown in Figure 14, except in reverse order.

After the correct sequence for change propagation is determined, a modal window is displayed, and the propagation of saves begins. Since modal windows block the processes in which they were spawned, the user is prevented from attempting to alter submodels while the changes and saves are propagating. Consequently, after propagation, model consistency is assured.

More specifically, after the propagation window is displayed, all of the modules in the interface list are saved to temporary files in the `.mobius/temp` directory. This save is done because if the change were canceled during propagation, then all of the changes that occurred as a result of propagation would also have to be canceled. The easiest way to do this is simply to restore the original state that all of the modules had before the propagation happened.

Next, `validateInterface()` is called on each interface in the list. This causes any module to become consistent with any changes in lower-level modules. If the validation changes are accepted, the next module is validated. If the validation changes are canceled, the original save is canceled. This ensures either that the changes propagate to all models necessary or that they are not saved in any model. Again, this is designed to prevent the user from generating inconsistent models. If all of the validations are accepted, then each module in the list is saved. Lastly, each module in the list is compiled, and the user is notified of any compilation errors in any model. Compilation errors do not mean that a model is necessarily invalid. More specifically, complicated functions within modules, like reward functions on performance variables or gate functions in SANs, may now have undefined state variable references, which the user needs to correct in order for the generated code files to be compilable.

At the end of the propagation process the window stays visible, but is no longer modal. Thus, the window may be used in correcting possible compilation errors while access is restored to other active windows in the current process. Any models containing the altered model are now completely consistent, although possibly not compilable, and the user is free to

continue editing his or her model. Because of this automatic propagation process, users do not need to worry about the complicated module dependencies, and changes in submodels, which are used in many places, may be performed quickly and efficiently.

4.2.5. Loading module interfaces for access

Before a particular module may access another, the interface of the module to be accessed must be "loaded." In Chapter 3, loaded modules were described as those without visible editors. However, the process of loading an interface involves more than deserializing the module or accessing the module in the control panel's cache. In particular, "loading" another module's interface also provides a means to prevent certain inconsistencies from occurring, specifically those that occur when multiple model constructors are being used. Loading accesses the open version of a module if there is one available. However, the open version of a module may contain unsaved changes. If the module that is accessed is not saved after it is loaded, the model may become inconsistent because, although the changes have been discarded, the accessing module may already have incorporated them. For example, this would occur if a user had unsaved changes in an atomic model, and then after the unsaved changes were incorprated into a composed model, the user closed the atomic model without saving.

In order to prevent these kinds of inconsistencies, a module's interface class should be loaded through the `loadInterface()` method before it is accessed. This loading process not only acts as a means to minimize module access time, but also ensures model consistency. Module loading provides fast access by deserializing a model only once, after which it is cached for future accesses. Consistency is ensured because the control panel will determine whether changes are unsaved in an open model. If there are unsaved changes, the user will be prompted to save them. This provides the user with a means for guaranteeing consistent models while being able to simultaneously edit different parts of the model.

Propagation guarantees that changes in all models are saved. Therefore, there is no need to prompt the user to save interfaces loaded during propagation. In order to handle situations such as propagation, the `loadInterface()` method detects whether a propagation is in process. If it is, then any open module's interface is automatically used

without prompting the user for approval. This ensures that propagation through many models, which may require loading numerous interfaces with unsaved changes, does not present a user with a stream of dialog windows to which he or she must respond.

4.2.6. Compiling

Compiling at the interface level is not done in order to create the final executable model. Instead, it is a means to ensure that any user specification results in the generation of valid C++ code for the model. For example, it is the means by which the user may find errors in the reward functions, the SAN gate functions, or any other user-defined code segment.

Typically, when a model is compiled, a window is displayed in which the messages to the standard out and error streams are displayed. If compilation is successful, the user is told so; otherwise, the user is presented with the errors. Just prior to compilation, the information class's `compile()` method is called. This method is expected to generate the required header and code files and a makefile for compiling them. (A more detailed description of this method is provided in Section 4.4.) Then the makefile is used to compile the model on the current architecture. Currently, g++ is the compiler used for this process.

There are several options enabled during compilation that deal with how the information is presented to the user. The compilation window may be displayed always, or only if there are errors. In addition, parameters allow a reference to the compilation window to be returned. This functionality is used during propagation, in which compilation windows are never shown; but the windows are stored in case the user wishes to see them. Since the propagation window is no longer modal after propagation, these compilation windows may be used as references during the correction of models with errors.

4.2.7. Importing

The importing functionality was introduced in Chapter 3, during the discussion on upgrades. *Importing* an interface refers to the process of taking an old interface class version and returning a new interface class version. The base interface class contains the method `importInterface()` for performing this upgrade function. An `object` is passed to this method since there may be many classes for which a new interface version is an upgrade.

Based on the class of the object that was passed, the proper upgrade process is undertaken to copy the old class's data into the new class's data structures. This process includes copying over file names, dependency tree information, and all model- and formalism-specific data. This method is empty at the base level, and should be overloaded by formalism designers. It is expected that these methods will grow with time as subsequent versions are released, to allow any existing models to be expressed in the most recent class version.

4.3. Editor Classes

The editor class is the second of the three base classes that form a module in the Möbius tool, and provides the fundamental means of specification in all modules. The base class for all editors in Möbius is the `Mobius.BaseClasses.BaseEditorClass`. This class contains many methods to enable designers of new modules to rapidly develop new derived editors which look and feel like other Möbius module editors. Further, the editor class also contains many expected system methods that, for example, interact with the control panel or their interface class. The main goal of this class is to make the derivation of new module editors for future research as simple a process as possible.

4.3.1. Undo functionality

One of the functions that the base editor supports is "undo functionality." *Undo functionality* provides a user with the ability to automatically return an editor to the state before the last change was made. The editor class uses two classes within the `Mobius.Utils` package to provide this functionality. These classes are the `UndoEvent` and the `UndoVector`.

The undo vector is composed of two associated vectors. The first is a list of undo events and the second is a list of descriptions of those events. The undo vector possesses methods to clear the lists and to add events and descriptions to the lists. The `UndoVector` class also implements a method to perform the first undo event on its stack. This method, `doFirstEvent()`, calls the `doAction()` method on the first event on the stack prior to removing it and its description from the undo vector's lists. Thus in order to make any user

event “undoable,” a derived undo event class should be created and its `doAction()` method should be overloaded to perform the actions necessary to counteract what the user did.

The base editor class implements this undo functionality through an `UndoList` data member. The undo list is derived from the `UndoVector` class. In addition, it is associated with the first menu item on the editor’s “Edit” menu such that the menu item’s text always displays the description of the next undo event on the list. Further, a user’s click on the menu item will call the `doFirstEvent()` on the undo list. Thus, in derived modules it is a straightforward process to determine how to undo an action. First, a derived undo event class that does the correct “undoing” actions must be created. Then that undo event is added to the editor’s undo list to make the action in a derived editor undoable.

This process is further simplified by the fact that all of the utility class visual components have default undo functionality already defined for them. This may consist of “unchecking” a checkbox or restoring the previous text in a text field. To enable this functionality in a particular Möbius utility classes, its `setUndoList()` method must be called after it is constructed. This method should be passed a reference to the editor’s undo list.

4.3.2. File menu commands

All derived editors also inherit their menu functionality from the base editor class. This functionality consists of typical File menu commands such as New, Open, Reopen, Save, SaveAs, Print, and Close. This section explains the functionality associated with these commands.

Two important editor methods are used frequently in these commands, `checkForSave()` and `checkForClose()`. The `checkForSave()` method first determines if there have been any changes in the model. Changes have been made if a special flag, `changed`, has been set or if any events have been performed such that there are undo events on the editor’s undo list. If there are changes in the editor, then the user is prompted to save if they so desire. A “Yes” response calls the Interface `save()` method to properly save the module. A “No” or “Cancel” response causes the module not to be saved.

In either case, if the module is not saved then the interface's `fixDependenciesInSavedModel()` method is called. This method was not described in the interface section because it is so closely tied to the editor. If the root list of the interface class has changed and the editor is being thrown away without saving changes, the saved model needs to be updated or the changes will be lost. If the root list has changed then there must be a serialized model file. The `fixDependenciesInSavedModel()` method opens this file on disk, updates its root list, and reserializes it. This may seem like quite a bit of unnecessary overhead, but it is necessary in order to maintain the consistency of a model's dependency tree. Canceling changes in a model's specification must not cancel the changes in a model's dependency tree.

The method `checkForClose()` detects if changes have been made, and if they have informs the user that they will be lost. If the user accepts their loss then the method returns `true`. It is not necessary for this method to call `fixDependenciesInSavedModel()` because it is only used when the user wishes to reopen the model, and the method for reopening is able to fix dependencies more efficiently than the more general fix dependencies call.

There is one other frequently used method in file menu commands. The `dispose()` method normally destroys a class and frees the memory associated with it for garbage collection. An editor's dispose method has been overloaded to interact more usefully with the Möbius tool, however. It hides the editor so that it is no longer visible. Then it notifies the control panel that the editor is no longer open. This enables the control panel to remove the corresponding interface from the open cache and place it in the loaded cache so that any future accesses do not require the model to be deserialized. Only if the model had never been saved, which would make loading of it by other models impossible, is the module truly destroyed.

The following list describes the editor's performance for file menu commands that a user may perform.

- **New**: When a user clicks the New file menu item, `checkForSave()` is called to allow the user a chance to save before their most recent changes are lost. Then a new instance of the module is created through the `Class` object's `newInstance()` method. After this the new interface's control panel reference

is set, and its editor is shown in the same place as the current one. Lastly, the current interface is disposed.

- **Open**: The open command first provides the user with a file dialog prompting the user to select a model file of the same type as the one from which the open command was issued. If the model specified was not the current one, it is opened (or accessed through the opened cached if it was already opened), and its editor is shown where the current one is. Lastly, the current interface is disposed.
- **Reopen**: Reopening a model will first call `checkForClose()`. If the user does elect to reopen, then most recently saved version of the interface is restored. This involves first deserializing the class from its associated file, and then invoking `fixDependenciesInSavedModel()`. Invoking this method preserves any possible new dependencies the interface may have after the model changes are lost.
- **Save**: When a user clicks on the save menu item, the interface's `save()` method is invoked. Two methods are then called before the module is serialized. Specifically, the `beforeSave()` method is for system use. It resets the `changed` flag, and removes all the elements from the undo list. Then, the editor's `save()` method is called. This method is empty in the `BaseEditorClass` because derived editors are expected to overload this method. It should be defined to perform any necessary data structure manipulation before serialization or compilation is attempted.
- **SaveAs**: The `saveAs()` command calls the interface's `saveAs()` method. Similar to editor saves, both the `beforeSave()` and the `save()` method are invoked on the editor.
- **Print**: The print command calls the editors `print()` method. At the base editor class, the print method is empty. Any derived module is expected to have this method overloaded to generate HTML code that properly documents the current editor specifications. The `Mobius.Documentor` package contains many classes that already generate HTML lists, images, and tables that module designers may use implement their formalism specific print functionality.

- **Close**: The close command simply calls `checkForSave()`, and then `dispose()` on the editor.

4.3.3. Textual editors

The simplest form of editor is the textual editor. This editor is still a graphical user interface, but the term *textual* here refers to the fact that a user is not required to draw as part of the module's specification. Instead, these editors are composed of a combination of graphical components within which the user is allowed to choose or type information as required. Figure 16 shows the editor associated with the Möbius simulator, as an example of a textual editor. It is composed of components such as choice boxes, text fields, and tab panels, from the `Mobius.utils` package.

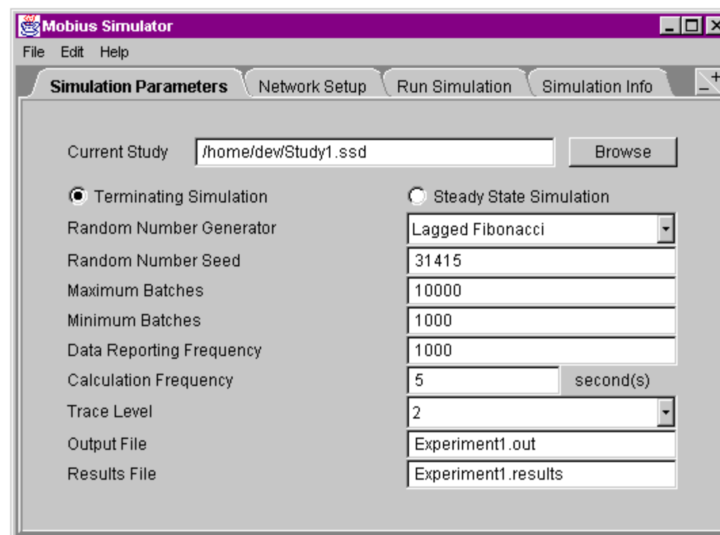

Figure 16: Möbius Simulator Textual Editor Example

4.3.4. Graphical editors

The graphical editor is more complicated to implement. The term *graphical* as used here refers to the fact that much of the model's specification is drawn by a user. There are many reasons why graphical editors are important. First, many formalisms, such as SANs,

Petri nets, and queuing networks, are naturally represented as pictures, and an editor for specifying pictorial formalisms should be graphical. Further, graphical representations of pictorial formalisms provide a modeler with a better way to visualize a model's structure than a textual equivalent would.

Implementing a graphical formalism from scratch would be a daunting task. To do so would require a great deal of overhead associated with creating basic drawing capabilities before any implementation of their module-specific functionality would be possible. Moreover, much of that work would be identical in different formalisms. Further, requiring all new editors to implement the same basic functionality individually would result in a different look and feel for every graphical editor. That would make learning new interfaces much more difficult. To avoid these problems, editors in Möbius may be equipped with a special panel in which all of the important formalism-independent graphical specification functionality has already been implemented.

The remainder of this section describes, in detail how this panel operates. It begins with an overview of functionality and initialization. Then the operation of the panel class is described. Specifically, the modes in which the panel may be operating are discussed. These modes are "create," "select," "selecting," "moving," and "connecting." Then, issues related to implementation and extensibility of the basic panel object are presented. That is followed by a description of two derived panel object types, vertices and edges. In particular, the way that these objects extend the base panel object, and may themselves be extended by formalism-specific objects, is described. During the discussion on edges, the three predefined edge types, which are straight lines, connected lines, and spline curves, are described.

Functional overview and initialization

Figure 17 shows the SAN editor, an example of a graphical formalism editor. The default panel functionality for graphical formalisms consists of graphs with directed edges and types of vertices. Objects representing formalism-specific notions may be placed in the panel as the vertices of the graph and connected by directed edges. One may move objects by dragging them, and select them by clicking on them. In the Möbius tool, an attempt was made to make interactions with these graphical formalisms through the panel as intuitive as

possible. Therefore, shift and control buttons are used for multiple object selection much as they are in Windows applications. Further, more complicated definitions about particular

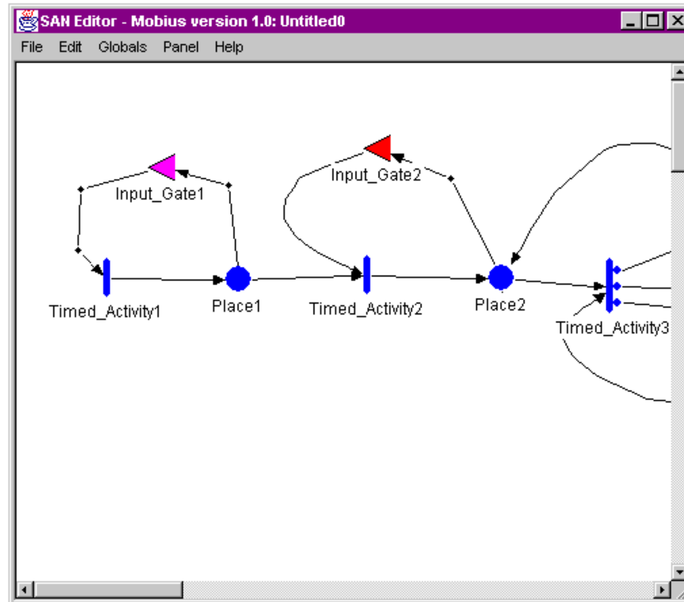

Figure 17: Möbius SAN Editor Graphical Editor Example

vertices in the graph may be specified through popup menus that appear in response to right-clicks on objects.

Typically, GUIs that contain the panel will do so within a scrolling panel. This scrolling panel is implemented within the `BaseScrollPanel` class, and implements horizontal and vertical scrollbars so that a small editor may contain a much larger panel. The traditional appearance of graphical editors in the Möbius tool contains a scrolling panel as the only component. The base scroll panel class is not abstract. This is because the scroll panel is an implementation of a simple container with scroll bars, and nothing formalism-specific is required for it to work correctly.

Certain methods must be invoked in an editor's constructor in order to notify it of the fact that it is graphical, and to enable graphical-editor-specific functionality. In order to enable panel interactions from the base class functionality, `DefineScrollPanel()` must be invoked and passed a reference to the scroll panel containing the desired formalism-specific panel class. Once the panel has been specified, an editor's `Initialize()` method should be called. First, this method calls several methods on the panel, which initialize the

panel with the information about the allowable formalism constructs. Then the `Initialize()` method adds to the editor's menu bar an additional "Panel" menu that is designed to interact with the panel. The next section consists of a more detailed discussion of this panel menu and the panel itself.

The panel class

Panels are classes designed to contain other components. In Möbius, the components they contain are canvases with pictures representing formalism-specific constructs. The panel is implemented in the `BasePanelClass`. As stated earlier, the panel is closely associated with a "Panel" menu on the menu bar. This panel menu contains a list of available formalism constructs and connectors that may be used to describe dependencies between those constructs. The formalism constructs are vertices in the connected graph, and the connections are edges. Another panel menu item is the cursor selection. The cursor selection is a means for interacting with the drawing, and it enables functionality such as multiple object selection. Figure 18 shows the panel menu for the SAN editor in the Möbius tool as an example.

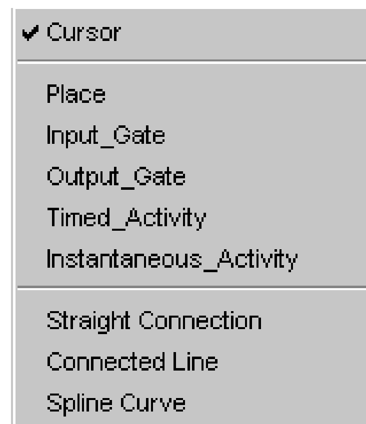

Figure 18: SAN Editor Panel Menu

The panel always operates in one of five modes: create, select, selecting, moving, or connecting. These modes are entered based on a combination of panel menu selections and user mouse events. Java's mouse listeners were used in order to catch any mouse events and detect the object that was their source. In determining the proper response, it is also important to know whether the event occurred within the panel, or within one of the objects the panel contains. The following list describes each of these operating modes in detail:

- **Creating:** *Creating* is the operating mode by which a formalism construct is added to the panel and consequently the structural model. This mode is enabled when the user selects a formalism construct on the panel menu. If the user clicks in the panel in this mode, an object of the appropriate type will be placed in the panel at the place that was clicked.
- **Select:** The *select* mode is entered when the mouse pointer is over a vertex in the panel, and the user has made either the cursor selection or a formalism construct selection in the panel menu. In this mode, clicking on the object will cause it to become selected. When selected, an object's picture may be altered. The panel keeps track of selected objects through a list called the `SelectedGroup`. When the mouse pointer exits the object, the panel returns to the creating operating mode. Holding down shift while clicking on the object will add the object to the selected group; if shift is not being held down, the object will become the only selected object.
- **Selecting:** The *selecting* mode, while similar to the select mode in that it is a way to select objects, is different in many ways. First, it is triggered by dragging the mouse in the panel. *Dragging* occurs when the mouse button is held down while it is moved. When this operating mode is entered, within the panel, a box appears whose corner is located at the mouse location, and which is resized as the mouse is dragged. If shift is held down during this operation, then any vertices that are contained within this box have their selection states toggled. If shift is not held down, then all vertices become deselected prior to the resizing of the box.
- **Moving:** The *moving* operating mode is entered when the mouse is dragged within a vertex. As soon as this mode is entered, a box appears that surrounds all the selected vertices in the panel. The box then moves with the mouse, and the selected vertices are then moved to wherever the user's drag ends.
- **Connecting:** The *connecting* operating mode is entered when a user selects an edge type from the panel menu. In this mode one begins, a connection by clicking on a vertex. After this initial click, additional clicks in the panel may or may not place edge points within the panel, depending on the type of edge selected. An

example of an edge point is the point at which a connected edge turns. The connection is terminated with a click on another vertex, and an edge of the appropriate type is placed in the panel connecting the two vertices. Right-clicking at any time after the initial click cancels the connection in progress.

In addition, undo functionality for interacting with some of these operating modes is already defined in the base editor class. The `BasePanelUndoEvent` class has been designed to allow for undoing object moves, connections, or group deletions automatically without a formalism designer needing to implement any of this functionality at the derived level. This class is derived from the undo event class, and the correct undo event is placed on the undo list as required for any user actions.

All operating modes are available within any graphical formalism. It is only necessary for the derived formalism-specific panel to maintain the information on what vertices are allowed. This information is communicated to the event-handling functionality and the panel menu through the method `RegisterObjects()` on the panel class, which is called during the editor's initialization.

All derived panels should overload the `RegisterObjects()` method to call `RegisterPanelObjectType()` for each type of formalism construct allowed. Because a derived panel object and its description is passed to each of these calls, the panel menu may be completely created with the correct formalism-specific construct descriptions when the editor is initialized. Further, each menu item on the panel menu may be associated with the correct type of object to create upon selection.

The panel object

There are several different kinds of panel objects, but all of them are derived from the `BasePanelObjectClass`. They are vertices, edge points, and edges. In this section, the implementation of the basic panel object class is described. Subsequent sections will discuss the implementation of the derived panel objects.

As stated earlier, the base panel object is a canvas that may display many types of picture. Panel objects may be in one of three states, and there are flags on the object for setting or determining that state. The possible states are highlighted, selected, or neither.

Objects are *highlighted* when the cursor is hovering over them in the panel. Objects are *selected* with clicks, and each group of selected objects may be interacted with as a single unit during moving or deleting operations. There are also flags indicating which states are allowable for a particular object. By setting these flags in derived objects, the functionality defined in the panel object may be activated or deactivated as desired.

The `paint()` method exists on all visual components in Java, including the `java.awt.Canvas` class from which the panel object is derived. This method is the method by which Java refreshes the appearance of all components. This method is overloaded on the panel object to call one of three methods, depending on its state. These methods are `DrawHighlighted()`, `DrawSelected()`, and `DrawSelf()`, and they are called when the object is highlighted, selected, or neither, respectively. In the panel object, these methods are empty since all graphical representations are formalism-specific. These methods must thus be overloaded for formalism-specific objects to correctly draw their graphical representations. Typically, the graphical representations for all states are identical except for the object's color, which may then be used to recognize the object's state.

Panel objects are also each named. A visible label displaying this name may be attached to the panel object. A flag indicates whether this label exists; if it does, it always appears directly below the panel object that it names. These labels were implemented in the `BasePanelObjectLabelClass` class. In addition, it is in the base panel object that undo functionality for any panel object is defined. Through the `BasePanelObjectUndoEvent` class, the creation, renaming, or deleting of any single panel object has been made "undoable" in a formalism-independent way.

Another important set of functions implemented by the base panel object is that of the popup interaction methods. As stated in the base panel section, one means for user interaction with formalism constructs is popup menus that appear when objects are right-clicked. Different popup menus may be defined for any derived panel object. These popups should be derived from the `BasePanelObjectPopupClass`, which by default allows for object interactions such as defining, renaming, deleting, and label hiding/showing. The `getPopup()` method by default returns one of these base panel object popups, but any derived object may overload this method to return whatever popup class is desired.

The means by which panel objects are created is also noteworthy. Recall that an object of each type must be passed to the panel through calls of the RegisterPanelObjectType() method. When the user wants to create a new panel object of a certain type, this original object's Duplicate() method is called. Duplicate() uses Java's predefined clone() method to create a second instance of the object. After the clone is created, the construct() method must be called. Cloning an object is a memory copy. As a result, string pointers and other object references point to the same place. It will create problems, however, because cloned objects do not refer to different addresses. For example, all the panel objects of any given type would point to the same location for their name, and changing that location would then change every name. The construct() method allows the specification of new addresses for each cloned instance. Derived panel objects should overload the construct() method to allocate space for any fields where these problems could be encountered.

When the construct() method is called, the object does not yet have a graphics object. A graphics object is the Java class for the storage of a visual component's picture. This means that some object definitions may not be performable in the construct() method. For example, determination of font information may not be performed until an object's corresponding graphics object has been instantiated. As a result, the Initialize() method is called directly after an object is placed in the panel. Any object definitions that must be performed after the graphics class has been instantiated may be performed through the overloading of this method.

Most likely, module designers will only need to specify vertices for their formalism constructs, since edges have been specified in a formalism-independent manner. Even if they do need to specify edges, these programmers will probably never need to derive any functionality directly from the BasePanelObjectClass, since more specific classes for vertices, edges, and edge points have already been defined in BaseVertexClass, BaseEdgeClass, and BaseEdgePointClass respectively. The following sections discuss each of these types of derived panel object in greater detail.

Vertices

Because significant functionality has already been defined in the base panel object class, the vertices, implemented in the `BaseVertexClass`, simply add interactions with connections. Since the structural models are directed graphs, vertex classes maintain lists of the inputs and outputs specified by their connecting edges. These lists are automatically maintained through the interaction of the base vertex with the base edge classes. However, specification of the rules for these connections between vertices has been simplified by a set of methods and data fields on the base vertex class.

All vertices have a `ValidConnections` list. This list is checked when any connection is attempted. This list will contain the descriptions of the valid output vertex types. These descriptions will be identical to the identification strings found on the "Panel" menu. In addition, flags allow module programmers to specify whether vertices have no valid outputs at all, or whether duplicate outputs are allowed. For the specification of more complicated connection rules, the method `validConnection()` may be overloaded to determine whether a connection is valid.

By default, any connection between vertices will appear to connect the middles of the vertices' pictures to each other. This functionality may be changed, though. In particular, a list of the points that specify where the user clicked to initiate or terminate the connection is maintained. Consequently, points other than the center of a vertex may be used as the endpoints of the connector. The endpoint of the connection is passed to the method `connectTo()` as specified by the user's click location, and the method returns the point where the connector's end point should be drawn. If this method is overloaded, a derived vertex may allow connections within a vertex wherever they are desired. This functionality is used, for example, in activities with cases in the SAN editor. The following section describes the implementation of `BaseEdgeClass`, the most basic connector object, and discusses the derivation of several useful edges.

Edges and edge points

The `BaseEdgeClass` has been implemented as a means for allowing rapid development of edges in the panel. Further, this class implements methods that interact with

vertices to maintain their input and output lists when they become connected. In addition, three types of edge have already been defined for use in any graphical formalism.

These predefined edges are the straight connection, the connected line, and the spline curve. Their menu items on the "Panel" menu are shown in Figure 18. A straight connection appears as a straight line between the two vertices. A connected line is a collection of straight lines that are attached and may turn at edge points. Lastly, spline curves are similar to Bezier curves except that they follow the contours of their defining points more closely. Figure 19 shows examples of each of these types of edge in Möbius's SAN editor. The classes `BaseStraightLineClass`, `BaseConnectedLineClass`, and `BaseSplineClass` are the implementations of each of these types of edges.

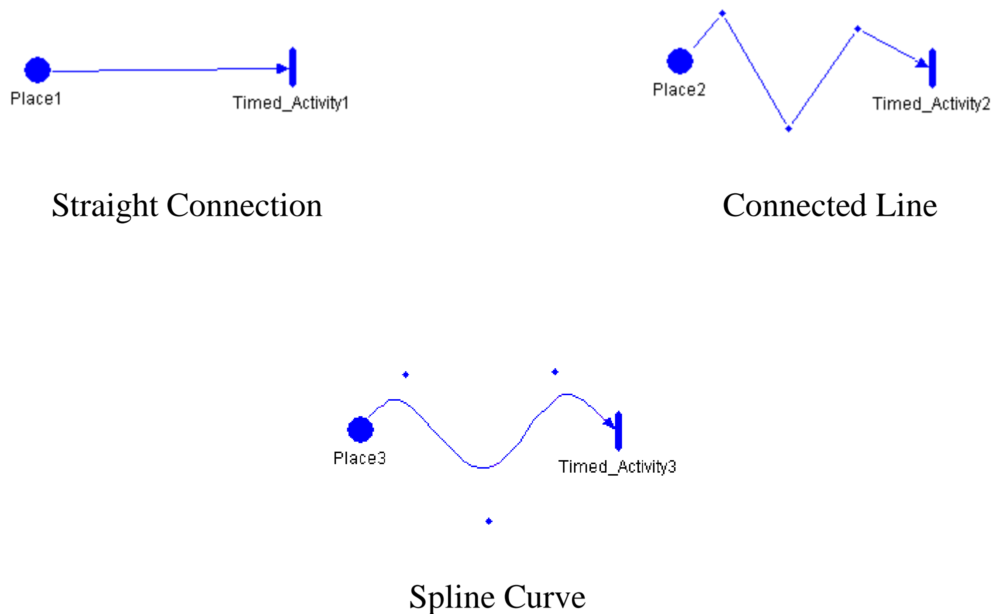

Figure 19: SAN Editor Edge Examples

In the case of the spline and connected line edges, the small diamonds along the edges are examples of edge points. For the connected line edge, the points are the places at which the straight lines that make up the connected line are connected. For the case of the spline curve, these edge points are used to calculate all of the points that actually lie on the spline curve that they define; therefore, they do not necessarily lie on the curve. These edge points are also extendible, such that different edges may have different-looking edge points. The most basic edge point is implemented in the `BaseEdgePointClass`. Further, the type of

edge point for any edge is specified by the edge's `produceEdgePoint()` method. This method must return an instance of the correct edge point class. If no edge points are allowed, for example in a straight connection, this method may return `null`.

All of these edges, although derived from the panel object class, are nonvisual. Instead, their appearance is generated using `BaseEdgePieceClass`. These edge pieces are capable of containing several points all connected by lines. The edges use the edge pieces rather than simply drawing the edge in a single canvas because all Java components are enclosed in rectangular boxes. That means that long edges are contained in large rectangular canvases. The larger these canvases are, the longer it takes to repaint them, and the greater the number of overlapping components. Overlapping canvases are a problem because the objects underneath are hidden. By using multiple edge pieces to draw an edge, the repainting speed is optimized. Edges may then implement algorithms such that a minimum number of these edge pieces is used to draw the edge. These components are flexible so that any new derived edges may also use them for their visual representation. The fact that straight lines, connected lines, and splines have all been implemented using these edge pieces is evidence of their flexibility.

4.3.5. Global variables

In addition to undo lists, file menu commands, and graphical panels, many formalisms may need to allow users to define global variables for models. Like global variables in programs, these variables in models are defined outside any particular part of a model. Thus, they allow information to be shared across the entire model. To make this easy, global variable declarations have been implemented in the `BaseEditorClass`. In the Möbius tool, studies allow ranges to be predefined on these variables. Consequently, the variables may be left undefined in the model itself, and then used as parameters later to easily alter a model's specification for comparison.

In order for an editor to create and provide access to global variables, its constructor must call the `enableGlobalDeclarations()` method. This method will add a "Globals" menu to the editor's menu bar. This menu contains two menu items. One menu item allows global variables to be declared, and the other allows the global declarations to be removed. Two lists on the editor, `localGlobalNamesList` and

`localGlobalTypesList`, maintain the names and types of any global variables that are declared in the current module.

The implementation of these global variables at the model specification level is simple. Any model specification in which global variables are used must simply declare an external variable of the correct type and name. Since a study is required on any solvable model in the Möbius tool, the study declares these global variables in a separate file. Model specifications are linked with these global variable declarations during compilation.

4.4. Information Classes

The information class implements communication between modules. In addition, it contains methods that the Möbius tool uses for the creation of executable models. The class that implements the information portion of the module is the `Mobius.BaseClasses.BaseInfoClass`. This section describes how compilation and communication are done in the Möbius tool to aid the designers of new modules.

4.4.1. Compilation

As said in Section 4.2.6, the `compile()` method on the information class is called prior to the compilation of the model. In the base information class, this method is empty, since the functionality associated with this method is completely formalism-specific. On each derived module, this method must generate C++ files that represent the structural model specified in the module's editor.

In addition, the `compile()` method must generate a makefile that compiles these files into an archive and place the archive in the correct architecture directory in the `.mobius/lib` directory. Generation of makefiles in the Möbius tool is done by the `MakeFileWriter` class. This class takes as input the list of model code files and the name of the archive to compile, and then generates a makefile that will work correctly within the Möbius file structure.

Because of the modularity of models, parent models must include the header files of the models they contain. Further, solvers require the names of every model's compiled archive so that they may be linked into a single solvable model. To support requirements such

as these, information classes must support methods to exchange information concerning the generated code and compiled archives for a model. To exchange compilation information, every derived information class must support five methods. In the `BaseInfoClass`, each of these methods is declared abstract; module designers must define them before the derived information class may be instantiated. Specifically, the methods are as follows:

- **`getListOfHeaderFiles()`**: This method must return a list of header files (.h or .hpp), with the path included, that are the C++ specification of the model.
- **`getListOfCodeFiles()`**: This method must return a list of code files (.cpp), with the path included, that are the C++ specification of the model.
- **`getMakefile()`**: This method must return the name of the makefile to execute in order to create an executable model.
- **`getArchiveFile()`**: This method must return the name of the archive into which the makefile places the executable model.
- **`getRequiredArchives()`**: This method must return the list of archive files that contain the formalism specification for the type of module.

4.4.2. Communication

The information that is communicated between modules, as well as the motivation behind the communication of that information, will be described in this section. All of the methods necessary for module communication are declared abstract in the `BaseInfoClass`. Like the compilation methods, module communication methods must be defined in each module.

In order to understand the methods, it is important to recall that there are three basic components of any model at the most abstract level: state variables, actions, and models. The information class for any module is required to support methods that return specific information concerning the state variables, state variable names, names of actions in the model, the list of submodels that a model contains, and the list of interface classes that correspond to those model names. Furthermore, any valid information class must support methods that return the names and types of the global variables declared in the submodel. These methods return a list of sets of global variable names or types respectively. The list is

indexed exactly like the list of models, and each set contains the global variable names or types that were defined in the corresponding model. In this way, all global variables declared in a model are determinable, as is the specific model in which they were declared. These methods are useful during the declaration of global variables within parent modules, since they ensure that all globals of a particular name are defined to be of the same type.

Specific types of information class within the tool may support other communication methods in addition to these. For example, the Möbius simulator accesses a list of performance variables in the Möbius performance-variable-model constructor in order to determine which are terminating and which are steady state variables. Such implementations may limit a module's flexibility, but may enable modules to work together more efficiently. Consequently, formalism designers must make sure not only that the abstract information methods are overloaded, but also that any methods required by potentially interacting modules are defined.

5. CONCLUSIONS AND FUTURE RESEARCH

This thesis has presented an architectural approach for developing modular and extensible applications, and showed how this approach was used to develop the Möbius performance/dependability evaluation tool. Within the tool, modules for both model construction and solution have been developed. Further, the model construction in the tool has been implemented using the Möbius framework's flexible notion of model specification. This enables models to be specified modularly, and allows solvers to act upon models without any specific knowledge of state. The core implementation of the Möbius tool developed in this thesis consisted of the implementation of the control panel, the base module functionality, and a library of additional supporting classes.

The control panel has been developed as the tool's main application. It allows new modules to be added or removed dynamically. Further, it provides a place from which any accessible module type may be launched. Modules are the tool's functional units for model construction or solution. Module implementation has been divided into three functionally distinct classes: the interface, editor, and information classes. So that new modules may be designed with minimum effort, basic versions of each of these classes have been implemented with much of the formalism- and solver-independent functionality. The base interface classes support module saving, opening, loading, and dependency maintenance. The base editor class implements undo and file menu functionality, and also provides extensible formalism-independent graphical interface objects. The information class specifies communication methods that modules must support which allow the exchange of both model- and compilation-specific information. In addition to those classes, supporting classes that correct some of Java's errors, provide platform independence, and allow for rapid GUI development have also been implemented.

The proof of how easy it is to develop modules has been shown through the implementation of a SAN editor, a replicate/join composer, a reward model constructor, set and range study editors, a distributed simulator, a state space generator, and several numerical solvers. All of these modules were developed in a short period of time once the Möbius tool architecture had been implemented. As a result, we are confident that it will be easy to extend

the Möbius tool using the results of new research. A few specific examples of this research that could enhance the work developed in this thesis include:

- The implementation of functionality to make graphical editors more useful, such as double buffering schemes that optimize panel refreshing speed and zoom capabilities for model visualization.
- The implementation of serialization methods such that saved model information may be minimal.
- The implementation of compilation and file structure support for non-UNIX systems.

Examples of research that could be used to enhance the tool in a more general way include:

- The development of more complicated connected models, not only to expand the scope of systems that may be modeled, but also to show the flexibility of the Möbius framework.
- The development of additional atomic formalisms in order to show that the Möbius framework allows for different notions of state to be shared.

APPENDIX: MÖBIUS TOOL FILE STRUCTURE

There are several types of file to discuss with respect to the Möbius tool. These are initialization files, configuration files, module data files, and executable model specification files. All of these files except the module data files are organized in a `.mobius` directory that exists in each user's home directory. The `.mobius` directory is organized as shown in Figure 20. It contains initialization files for specification of startup parameters, a `temp` directory for storing useful information during execution, and a `lib` directory that is designed for architecture-specific executable model specifications. Within the `lib` directory, a directory will exist for each architecture on which a model has been compiled. The architecture directory, in turn, contains a compiled archive file, which contains an executable model compiled for that architecture.

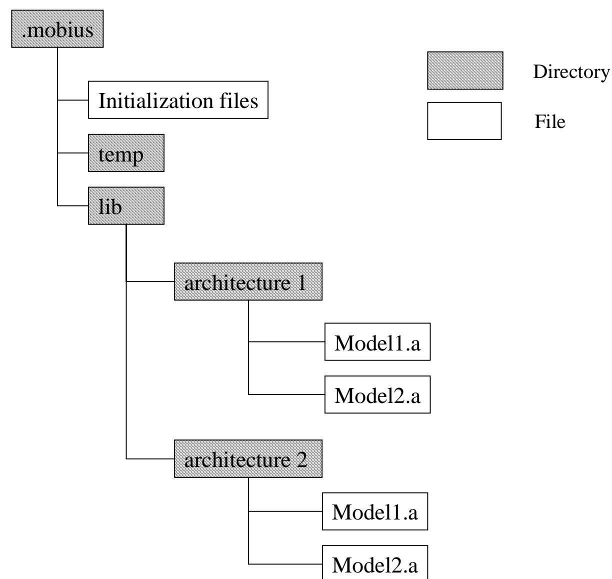

Figure 20: Möbius Tool File Structure

Module data files contain structural descriptions of model-specific information, as well as information concerning model dependencies and information regarding the files necessary for model compilation. In addition, these files are stored in a module-specific format. These files may be stored anyplace where a user has read/write access within the file system, and it is the user's responsibility to organize them efficiently.

REFERENCES

[1] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[2] W. H. Sanders, "Construction and solution of performability models based on stochastic activity networks," Ph.D. dissertation, The University of Michigan, Ann Arbor, Michigan, 1988.

[3] *UltraSAN User's Manual Version 3.0*, University of Illinois, 1995.

[4] F. Bause, "Queueing Petri nets: A formalism for the combined qualitative and quantitative analysis of systems," in *Fifth International Workshop on Petri Nets and Performance Models*, Toulouse, France, October 1993, pp. 14-23.

[5] R. A. Sahner, K. S. Trivedi, and A. Puliafito, *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package*. Boston: Kluwer Academic Publishers, 1996.

[6] D. D. Deavours, "The Möbius framework," unpublished, in progress.

[7] W. D. Obal II, "Measure-adaptive state-space construction methods," Ph.D. dissertation, The University of Arizona, August 1998.

[8] J. M. Doyle, "Development of the Möbius abstract model specification," M.S. thesis, University of Illinois, Urbana, IL, in progress.

[9] J. Sowder, "State space generation techniques in the Möbius modeling framework," M.S. thesis, University of Illinois, Urbana, IL, 1998.

[10] A. Williamson, "Discrete event simulation in the Möbius modeling framework," M.S. thesis, University of Illinois, Urbana, IL, 1998.

[11] D. Flanagan, *Java in a Nutshell: A Desktop Quick Reference*, 2nd ed. Sebastopol, CA: O'Reilly & Associates Inc., 1997.