A VERSION CONTROL INTERFACE FOR GRAPHICAL DISCRETE-EVENT MODELS

BY

DAVID HUANG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Adviser:

Professor William H. Sanders

ABSTRACT

Möbius is a discrete event modeling tool used for the creation and analysis of complex system models. Such models are typically graphical in nature and are built and maintained using editors provided with the tool. Typical project environments strongly resemble those of conventional software development, as they both exhibit similar problems such as project complexity and size scale-up. Specific problems include difficulty sharing work between team members, confusing and error-prone workflows during resolution of conflicting changesets, and the need for additional processes to manage multiple versions of the project or project components. Those problems are handled in the software world via version control systems (VCS): sets of tools and procedures designed to automate the more tedious parts of the problem where possible and to ease the burden on human developers when their input is needed. However, such solutions are designed for text, making them difficult to use with a Möbius project.

In a normal workflow, model developers use an internal Möbius editor to view and modify the models that make up a project. The models are saved to disk as formatted text files not intended for direct human use. While they encode all the features needed to reconstruct the model, a model developer directly viewing the text would have great difficulty parsing the higher-level meaning of the model. A common task when maintaining a large software project is that of comparing two versions of a file, often to resolve conflicts between more changesets or to triage a newly discovered bug. When performed on the raw textual form of a model, that task is difficult and very prone to error, since the high-level graphical concepts do not map cleanly to independent text snippets.

This thesis presents a structure with which a VCS could be integrated into Möbius projects. The proposed system includes direct integration of core VCS features directly into the Möbius user interface as well as a set of changes to Möbius graphical editors to allow visual comparison and merging of models. Furthermore, the proposed system is implemented using the Git VCS and the ADVISE model formalism as a proof of concept.

To the family and friends who believed in me when I did not.

ACKNOWLEDGMENTS

I would like to thank my adviser, Professor William H. Sanders, for his guidance and unwavering support, and for the opportunity to work and learn as part of the PERFORM group.

I would like to thank Ken Keefe for his technical insight and infinite patience. I would also like to thank Dr. Brett Feddersen for his support, advice, and numerous illuminating discussions.

I would like to thank Michael Rausch and Ronald Wright for their contributing work on ADVISE, and the other members of the PERFORM group: Atul Bohara, Carmen Cheh, Ahmed Fawaz, Mohammad Noureddine, Uttam Thakore, Benjamin Ujcich, and Varun Krishna. I also greatly appreciate the editorial feedback provided by Jenny Applequist.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

ADVISE	ADversary VIew Security Evaluation
AEG	Attack Execution Graph
AFI	Abstract Functional Interface
CLI	Command Line Interface
EMF	Eclipse Modeling Framework
GEF	Graphical Editing Framework
GUI	Graphical User Interface
IDE	Integrated Development Environment
MVC	Model-View-Controller
SAN	Stochastic Activity Network
UI	User Interface
UX	User Experience
VCS	Version Control System
XML	Extensible Markup Language

CHAPTER 1

INTRODUCTION

1.1 Motivation

Möbius [1], [2], [3] is a software tool used to construct and analyze discrete-event models of complex systems, typically in order to study reliability, security, or performance. The tool recognizes a number of model formalisms from which a project can be constructed and includes purpose-built editors for each formalism with which users can create and modify model data, in much the same way that software frameworks and libraries are used in the creation of larger software projects. Möbius models can also be numerically solved for steady-state system behavior or directly simulated when quantitative solutions are not possible or practical.

The development workflows and lifecycles of larger models strongly resemble those of software projects but have historically lacked native toolchain support for modern software engineering processes [4], such as regular automated testing, live debugging and analysis tools, and versioning systems. These problems are not limited to Möbius, but affect model-driven development as a whole [5]. In the long term, the situation reduces work efficiency across teams and reduces maintainability of models. There has been steady improvement of Möbius in recent years, with the additions of a debugger [6] and a shell interface to enable automation [7]. We would like to continue that trend by providing native versioning support to further aid developers in the future.

In the realm of software engineering, *version control* refers to the explicit storage, management, and tracking of the history of changes made to the source code of a project. There are many benefits:

1. Access to detailed changelogs of a project over its lifetime.

2. Improved collaboration via systematic branch and merge, allowing multiple developers to work on the same files while minimizing conflict.

3. Atomic changes (i.e. all or nothing) to keep related files synchronized with each other.

4. The ability to roll back to a previous version if/when critical problems are discovered.

The problems addressed by implementing version control into a conventional software project are also present in large Möbius projects. Point 3 in the above list is particularly relevant due to the rigid definitions of each model formalism. Copying or filtering through changes by hand is prone to error and may thus easily result in a nonsensical or otherwise invalid model. Furthermore, most formalisms in Möbius are graphical in nature and difficult to work with in a textual format, making native VCS integration even more attractive.

1.2 Thesis Overview

This thesis presents an extension of the existing Java backend and user-facing editor code to formally support version control in the Möbius tool. While the high-level methodology is formalism-agnostic, the currently supported formalisms vary in their internal implementations. In the interest of time, the presented methodology was applied to the ADversary VIew Security Evaluation (ADVISE) formalism [8], [9], but efforts were made to keep high-level abstractions out of the formalism-specific source code. The implementation described in this thesis requires models that are built using the Eclipse Modeling Framework (EMF) and Eclipse Graphical Editing Framework (GEF). It is expected that in future Möbius development efforts, all formalisms will be migrated to those frameworks.

A great variety of version control software tools are available, each with its own workflow chains and change models. Because of its widespread popularity and ease of use, Git was chosen for the implementation presented here. However, the changes to the Möbius codebase were implemented in such a way as to minimize the amount of additional work needed for a drop-in replacement VCS.

The goals of this project are to enable the following capabilities in Möbius:

1. Place a project under version control with awareness of the structure of a Möbius project.

2. Allow a user to access basic VCS operations (such as log, pull, and push/commit) across a project from the Möbius graphical user interface (GUI).

3. Support visual version comparison on arbitrary project files via a textual comparison tool.

4. Improve the graphical model editor to allow visual comparison and merging.

5. Ease further development of Möbius by designing the above changes to be compatible with future feature updates.

The above changes will allow a user to manage changesets from within the Möbius tool. That will minimize the risk of introducing unintended behavior when one is reviewing or making changes to a project. It will also provide tools for reverting to a previous working version if a problem arises.

1.3 Thesis Organization

The remainder of this thesis discusses the design required to enable the above features as well as the structure of the changes made to Möbius in implementing those features. Chapter 2 describes a few of the intricacies of the relevant software frameworks and what changes may be necessary when implementing our design for a different formalism or VCS. It also briefly explores other work done in the area of graphical comparison tools. Chapter 3 details the specific new features that an end user may use as a result of this work. Chapter 4 discusses the implementation details involved in using EMF and GEF to add these features to ADVISE. Chapter 5 includes a brief conclusion on the work done as well as possible future work or extensions. Finally, Appendix A offers low-level details on how to extend this work for other formalisms within Möbius.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Version Control Software

Currently, there exist a large number of VCS, both free and proprietary. The specific VCS used for this project is largely irrelevant, but the methods by which changes and workflows are modeled in each of these systems vary quite drastically. Operations or concepts that are common in one system may not be applicable in another, making it impossible to create a unified user interface for all VCS systems while maintaining enough fine-grained control for meaningful user interactions. For example, the concepts of local clients and file view mappings in Perforce are nonexistent in most other systems [10]. The feature must either be abstracted away and hidden from the user (limiting the expressiveness of the user-facing VCS commands available in Möbius when Perforce is used) or be disabled when a project is found to be using a different VCS that lacks the feature in question. As a result, each VCS requires a unique implementation within Möbius to account for its individual quirks; there is no "one size fits all" solution. Indeed, VCS support in the integrated development environments (IDE) used for software development is implemented separately for each unique system.

A significant amount of technical infrastructure is required to support a VCS capable of more than simple local versioning, so the specific system used depends largely on what the end user or team is familiar with and what is readily available to them. For this project, Git [11] was chosen as an example implementation to demonstrate what is possible. Git is a decentralized VCS and a reasonable choice for the pilot implementation of this Möbius feature, primarily because of its popularity, which will increase the likelihood that a given model development team will be familiar with the tool and already have the necessary infrastructure in place to support its use. Other tangential benefits include Git's distributed nature (i.e. working on a local model is the same as working on the "master" copy, which decouples the Möbius VCS feature from a network dependency) and its support for more advanced VCS workflows, such as branching and tagging, which may be leveraged for future improvements to the Möbius VCS feature (see Chapter 5).

2.2 Möbius

A typical Möbius model development workflow is roughly divided into five parts (see Figure 2.1).

1. Construct atomic models: *Atomic models* are the most fundamental models available in Möbius and consist of a collection of state variables and events. Development workflows begin by creating one or more atomic models using one of the formalisms supported by Möbius.

2. Compose models: Multiple atomic models can be *composed* into a single model in order to facilitate the creation of complex system models from collections of simpler parts. Model composition is not necessary if the atomic models are sufficient to define the system of interest.

3. Define reward variables: After a system model is created, *reward variables* must be set to identify the system metrics of interest.

4. Set study parameters: A Möbius *study* contains the user desired values of any system parameters defined in the model. Multiple sets of values can be defined within a single study.

5. Solve/simulate: The model is combined with the configurations defined by the reward variables and study to obtain the steady-state of the system through numerical evaluation or simulation.

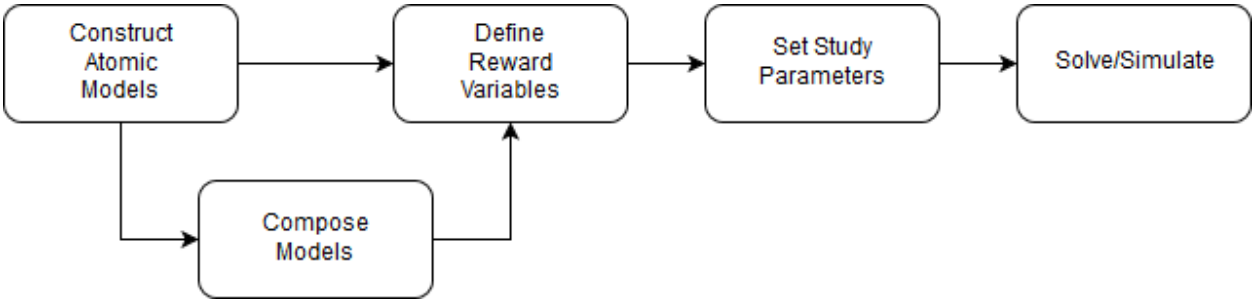

Figure 2.1: Möbius model development pipeline.

The Möbius model formalisms described above are the primary focus of this thesis work; they are internally implemented in the back-end using the Möbius abstract functional interface (AFI) [12]. It is a layer of abstraction that is shared by all model formalisms and implements their lower-level commonalities, such as model states and transition functionalities. It is this layer that enables the extensibility of Möbius, allowing one to add new

formalisms without needing to implement them from the ground up, even for such disparate formalisms as SANs and ADVISE.

While it would be advantageous to implement model version control at the AFI level for the purposes of code reuse between model formalisms, the AFI is at too low an abstraction level for this feature to be implemented in a useful manner. The AFI is excellent at expressing the mathematical similarities between Möbius formalisms but is difficult to use directly. Indeed, that is the precise reason why different high-level formalisms have been developed in the first place: while they are all equal in expressive power, it is generally easier to design and build a model in a higher-order formalism for most problems. There is again an analogy to software engineering, with which development times shorten as one moves up the abstraction stack of programming languages by offloading a greater portion of low-level implementation details to be handled by the computer. In addition, some formalisms have been designed for specific purposes, such as ADVISE for the security domain. Those abstractions are reflected in the structures of each higher-order formalism. If the VCS feature were implemented at the AFI level, it would be necessary to map detected differences back up to the layer of each individual formalism in order to present them to the user. The situation is analogous to that of source code written in C++ at the assembly level: difficult, error-prone, and nondeterministic when mapped back up to the higher layer. Support for new formalisms would also necessitate a great deal of additional work for each to enable new features. Instead, model comparisons will be made at the model formalism layer (see the discussion of EMF in Section 2.4), where the higher-order abstractions live.

Unfortunately, as a result of the above difficulties, a nontrivial subset of newly added features must necessarily be implemented on a case-by-case basis for each formalism. In general, such features are related to the specifics of each formalism's graphical editor and how each one displays the various elements and constituent parameters that make up a model, with particular focus on the unique display features of each formalism, if any. The reason is that, at the time of this writing, very little is shared between graphical model elements across formalisms, since internal implementations vary significantly. The situation may change in the future, and details on how the VCS feature could be either extended to fit current formalisms or adapted for an updated back-end are included in Appendix A.

2.3 ADVISE

As the name suggests, the ADversary VIew Security Evaluation (ADVISE) formalism [8] [9] was originally designed as a security-focused alternative to traditional system modeling

formalisms. This thesis is not an exploration of ADVISE (or any other formalism), but a brief background on ADVISE is needed to understand the implementation details of this work, since ADVISE is used as the vehicle for visualizing the changes within Möbius.

An ADVISE model consists of two parts: the secure system in question and a profile of the adversary threatening this system. The former is represented by an attack execution graph (AEG) and will be the primary focus of the implementation section of this thesis, as the adversary profile is not presented as a graph. Most nodes in an AEG represent resources relevant to the security of a system. Some examples of such resources include keys (both physical and digital), knowledge of internal processes or system architecture, and the adversary's expertise with a technology or technique. Together, these are represented in the AEG by the Access, Knowledge, and Skill nodes, as shown in Figure 2.2. Goal nodes broadly represent potential objectives of an adversary, from service disruption to data exfiltration. These nodes may also represent subgoals an adversary may attempt to pursue as steps towards a greater objective. Finally, attack Step nodes represent actions an adversary may take by utilizing the resources he or she controls to achieve one or more Goals.

Figure 2.2: ADVISE node types.

In an ADVISE model, the resources available to the adversary and the status of each goal in the AEG are abstracted as state variables. Model analysis consists of exploring the possible attack steps available to an adversary and predicting the likelihoods of various attack vectors based on the strength of the modeled adversary. These attack steps drive changes in the state variables and are defined by a number of parameters, including a *timing distribution*, that describe the rate of the stochastic process used to model the attack step, as well as a set of possible *outcomes* expressed in C code and their associated probabilities.

Nodes in the AEG are connected by directed edges and show the preconditions necessary for attaining the resource represented by each node. For example, incoming edges on a Step node indicate that the resources from which those edges originate are needed before that attack Step may be attempted. Similarly, incoming edges on an Access node indicate what is needed to obtain that Access resource.

Finally, of particular relevance for the graph comparison work in this thesis, most nodes

have attributes that are not visually represented on the AEG. The most basic of such attributes is the node name: a string that describes what resource or action the node represents. Other examples include the *timing distribution* and *outcome* attributes described above. Because changes to these nonvisual attributes are nontrivial and can lead to dramatically different model behavior during analysis, they must be accounted for when one is visualizing the resulting differences after a model comparison.

2.4 Eclipse Modeling Framework (EMF)

Earlier development in Möbius was primarily done using custom C++ code to implement the back-end solvers and formalism libraries. Front-end model editors and other GUIs were built with custom Java code [2]. While the Möbius AFI offers a useful mechanism for reusing the computational commonalities of various formalisms, a significant initial cost is still needed to define new ones. Therefore, the plan is to build future formalisms with EMF, and to port existing ones to this framework.

EMF is a modeling framework and scaffolding tool used to support and generate code based on a structured data model [13]. Broadly speaking, graphical formalisms can be simply described as a collection of node classes, each with a set of attributes and methods or events. For example, the node classes of a stochastic activity network (SAN) consist of places, timed and instantaneous activities, and input and output gates [14]; node attributes include gate predicates, activity cases, and place markings; and, of course, activity effects are good examples of node events. The above description of defined SAN components can be described using EMF's Ecore format, which allows large parts of a Java representation and front-end code to be implemented with relatively little effort. The built-in Ecore editor may also be used to generate boilerplate code that would otherwise be tedious to write and maintain. The adoption of this framework does not affect back-end performance, which is still implemented as a C++ compiled executable and a layer is still needed to compile models built with a Java-based editor to the models' corresponding C++ source code.

2.5 Graphical Editing Framework (GEF)

Just as the Java model representation in Möbius is migrating to EMF, the Java front-end presentation is migrating to GEF [15], a library specifically designed for the implementation of graphical editors. Previous work requiring custom editors for each formalism was cumbersome and costly. Implementation of such editors with GEF will enable greater code reuse,

both among Möbius model formalisms and with the wider GEF-using community.

GEF follows the standard Model-View-Controller (MVC) pattern so often used in UI design. Under that architectural pattern, clear distinctions are made between the "business logic," user interface, and user-visible graphics portions of an application. In particular, GEF provides a standardized model diagram editor, along with implementations of graphical components to represent model elements and infrastructure to support user interaction. The majority of GEF features are not relevant to the work presented in this thesis. Only small portions of the Möbius-specific implementation are modified in this work (for the purposes of slightly changing the presentation of a model during a comparison operation) and GEF's complex architecture is far beyond the scope of this work. However, the implementation presented in Chapter 4 is entirely dependent on GEF and the feature set it provides.

2.6 Related Work

Most of the related work done in the past concerns the mathematical and algorithmic details surrounding the problem of graph comparison as a subset of the greater study of graph theory. There has also been past work surrounding the visual presentation of graph comparison, but it has been comparatively limited, likely due to the user-driven and use-case-specific nature of user experience (UX) design.

2.6.1 Graph Comparison

While nongraphical model formalisms exist and are supported by Möbius, such as the Performance Evaluation Process Algebra (PEPA) formalism [16], they are beyond the scope of this thesis. Comparison and merging of such formalisms are already well-supported by conventional tools intended for text. Although such tools fail to capture the semantics of nongraphical formalisms, the outputs of these tools are fairly readable and useful, as the input model formats are already in forms intended for human consumption. If those tools are combined with Möbius (or similar software) in order to check model validity after a compare and merge operation, the result is adequate for most comparison-related operations.

For graphical formalisms, each formalism can be generalized to a directed graph, which is a collection of vertices connected by directed edges. Each vertex and edge may additionally carry some other data or property. In reference to the description of a SAN in Section 2.4, the node type (gate, activity, etc.), place markings, and gate predicates are all examples of such additional data. For the purposes of this work, such additions are assumed to be small

compared to the larger structure of the formalism. While there is nothing preventing a model developer from writing hundreds of lines of code for an activity case, doing so will generally go against the intended use case of the formalism as the logic is likely better expressed using additional graph elements. Transitioning code to graph elements eases human development time and effort, and allows the Möbius program to make optimizations. Thus, this thesis will more greatly emphasize comparison of graph elements rather than textual elements such as might be found in a model's vertex or edge attributes.

The theory of graph comparison algorithms is lengthy, and a large body of literature exists to study both the broader topic and related subproblems with much greater mathematical and algorithmic rigor than is relevant for this thesis. A succinct summary of some related work is presented in [17]. Of particular interest are the sections concerning subgraph matching and graph isomorphism problems. To be specific, *isomorphism* refers to the "structural equivalence" of two graphs: they contain the same set of nodes, and the edges in the two graphs are connected in equivalent ways. Solutions for isomorphism and subgraph matching problems aim to detect this equivalence either between two graphs or in subgraphs thereof. To generalize the VCS feature to other modeling tools and formalisms, it is necessary to implement some subset of algorithms to solve the isomorphism problem, either directly or indirectly. Luckily, a number of assumptions can be made when implementing the VCS feature (such as an assumption that relatively small changes will occur in the local neighborhoods of each graph element) to improve both the runtime and development costs of such implementations. In this thesis, such graph comparison algorithms are provided by an open-source library implementing a graph similarity algorithm. That library will be discussed in more detail in Section 4.2.

2.6.2 Visual Presentation

After comparison data have been collected and processed, it is still necessary to present the data to the user in a meaningful manner. While much research (and derivative implementations) has been done on the theoretical portion of the former task, there has been less work on the latter because of the highly use-case- and implementation-specific nature of the problem. In general, the goal is to display the comparison result so as to:

1. Emphasize elements of a model that have changed between versions and deemphasize those that have not.

2. Make clear what changes were made on a modified element.

3. Allow the user to accept or revert each change individually.

4. Allow the user to add other changes if the changeset presented is insufficient.

Those goals have been achieved in most modern textual comparison tools, which typically present the differences as highlights while showing the compared strings side by side. To support item #3 above, user interface (UI) elements are added to "push" changes from one version to the other. Finally, in the interest of item #4, they are also functional text editors. The behavior of such tools can be erratic and unexpected when large changes are compared or when the compared files are not related, often resulting in either the detection of similarities by sheer coincidence or large, unhelpful blocks of detected changes. That is a general weakness of all comparison operations and will not be addressed in this thesis.

A few methods by which graphical differences may be visualized can be seen in [18] and [19]. One option, which is similar to the method used by textual comparison tools, is simply to present the two models side by side. Ideally, scrolling should likewise be kept synchronized, both horizontally and vertically. That generally works well and presents a clear picture of the changes to the user, but can be confusing to use when one is attempting to produce a merged model, as matched differences may be visually spread out, and deleted elements do not appear at all when one is viewing the current version of the compared model. Again, if we borrow from the realm of textual comparison, those problems may be somewhat alleviated if differences are highlighted, with different colors being used to indicate how elements have changed. Another option is to merge the two compared versions together temporarily and present them together as a single model, with changes highlighted. That is complex and difficult to use when one is determining the previous state of a particular change, but it is more usable for merge operations, as the merge is already done and user input is required only to accept or reject each change. More exotic alternative options exist as well, such as animating the transition from one version to the other, or using contextual popups to navigate the changeset, but these less traditional presentations are difficult to use without significant user training. More research would likely be needed to expand those alternatives into usable prototypes.

In order to maximally borrow from the existing Möbius graphical model editor for each supported formalism, the merged method was chosen for this implementation. That choice preserves the familiarity of the primary tool used by Möbius model developers and allows for greater code reuse between the normal editor and that used for model comparison. Furthermore, it better facilitates the examination of nonvisual changes as described in Section 2.3 by displaying changes as an enumerated list overlaid on top of a merged preview model rather than as side by side comparisons of what would otherwise appear to be identical models.

CHAPTER 3

FEATURES

3.1 Basic VCS Operations

IDEs often offer a basic set of supported VCS operations that are either built into the tool or available as add-ons. They are rarely as powerful or expressive as the command line interface (CLI) and are not intended to be a complete replacement. Instead, they give users a way to perform the most commonly used actions without leaving the editor, and allow the editor to take control of versioning for project management purposes. Similarly, the new VCS feature in Möbius provides access to a number of operations available in the main UI. These operations are available as a right-click context menu on any project root and support the following (see Figure 3.1):

1. Status: Display current status of the project compared to the previous version, shown as a list of added, deleted, or modified files.

2. Log: Display a brief summary of the past few commit descriptions.

3. Diff: Perform and show the results of a project-wide textual file comparison.

4. Commit: Bundle up local changes as a single change and submit to the VCS for tracking along with a user-entered description.

5. Push/Pull: Update either the local or remote repository with the most up-to-date changes of the other.

6. Reset: Discard locally made changes since the last VCS commit.

Additionally, a **Compare...** contextual option is also available on any project component. For most components, this simply launches an external text comparison tool, as shown in Figure 3.2. For graphical components that support the newly added comparison and merge features, it runs the comparison algorithm and displays the result, as detailed in Section 3.2.

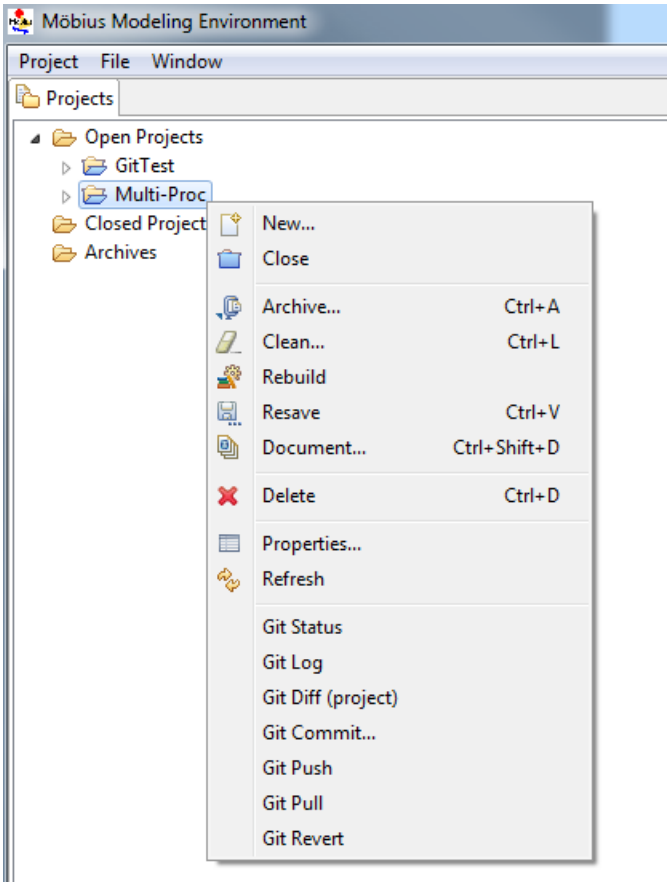

Figure 3.1: Project context menu with new options for VCS operations.

Of course, none of those features are functional without an initialized VCS in the project directory. When one is creating a new project in Möbius, one can use a new option we have provided to enable version control for the project (see Figure 3.3). For projects that utilize this option, the local repository can optionally be initialized to point to a remote server for group development efforts. Switches are available for fine-tuning the set of file types to be managed by the newly created repository. That allows users to opt out of version control on generated files of their choosing, which can dramatically reduce the size of the repository and improve performance when source files are being shared with other developers or when a local copy is being updated from a remote server, at the cost of increased compile time (as the intermediate files need to be created locally instead of being fetched from source control).

As mentioned in Section 1.2, the implementation presented in this thesis uses Git as the back-end VCS and consequently uses Git terminology in the operation names. However, the feature has been designed to allow other VCS to be integrated with minimal additional development efforts. For details, see Appendix A.

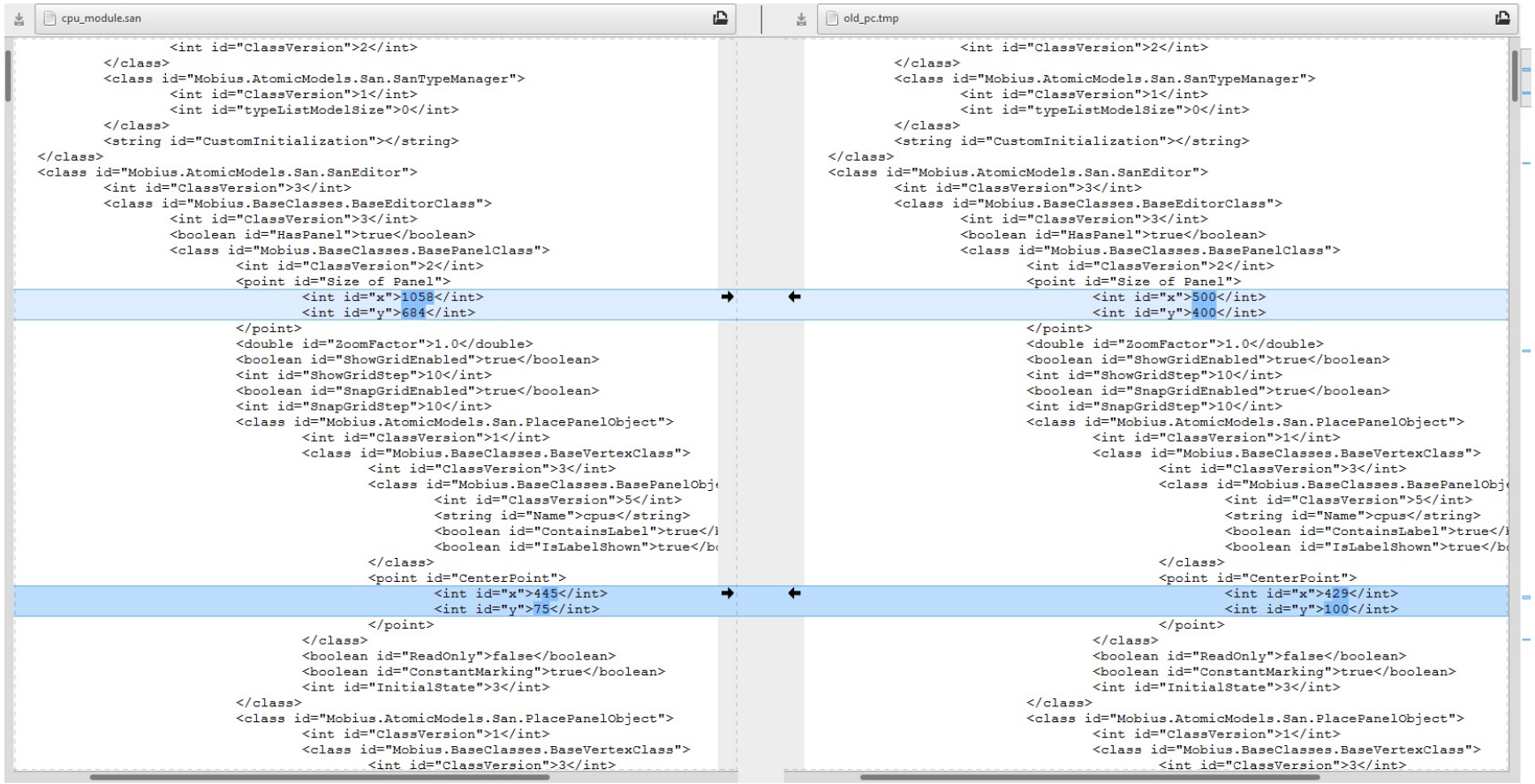

Figure 3.2: Meld text comparison tool, showing minor changes on a SAN model.

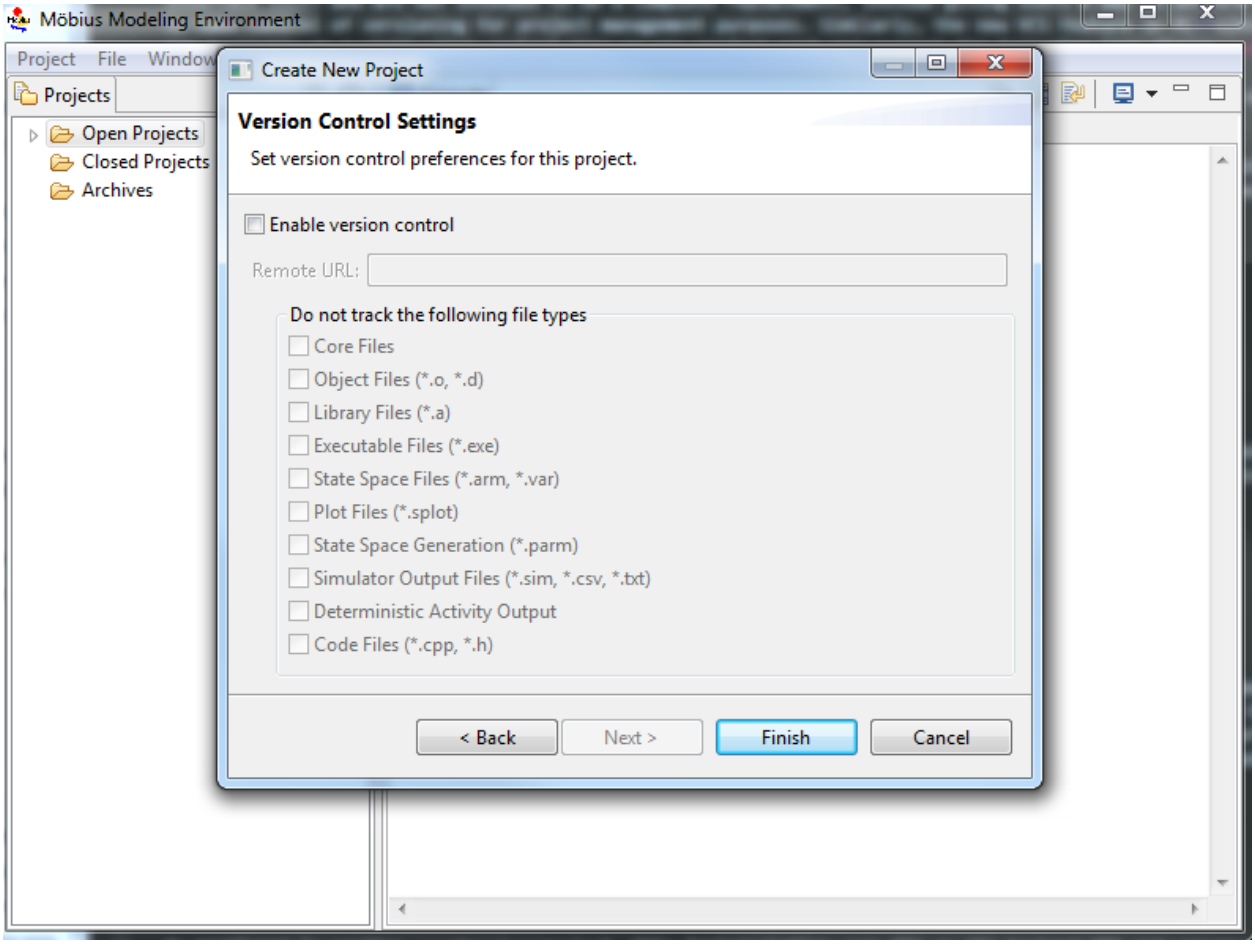

Figure 3.3: New project page with Git initialization options.

3.2 Graphical Comparison

Möbius models are stored as Extensible Markup Language (XML) files on disk. While these files are readable by humans, even a modestly sized model of about a dozen nodes with connections can be very difficult to understand in its textual form (see example in Figures 3.4 and 3.5). For that reason, the version control feature includes a graphical comparison (or diff) option, which allows the user to visually compare two similar models or two versions of the same model within the Möbius graphical editor for the associated formalism. The current implementation only supports models built using the ADVISE formalism, but we plan to add support for other formalisms as older formalisms are updated to use newer back-end libraries.

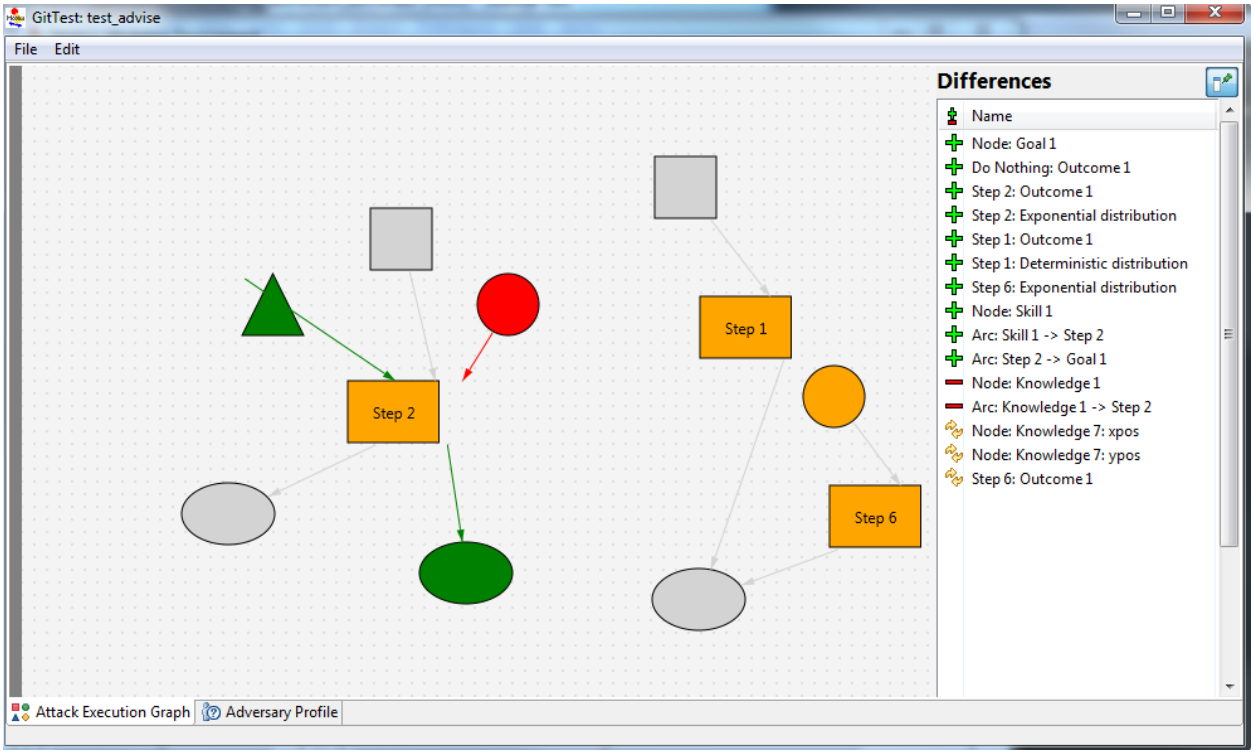

Figure 3.4: ADVISE AEG editor in Diff Mode.

3.3 Merge

In addition to the normal editing functionalities of the Möbius graphical editor, the version control feature enables users to accept or reject individual changes. In a typical workflow, version comparisons are often performed either to triage a discovered bug or to resolve conflicting changes made by multiple developers. In both cases, assuming that neither commit/changelist under comparison can be accepted verbatim, the next step is to cherry-pick the select changes to be discarded and accept the remainder. In the Möbius graphical editor, each change is displayed in a pull-out panel and can be either accepted or rejected via the right-click context menu attached to each change.

3.3.1 Merge Error Checking

As an addition to the merge feature, new logic has been added to catch potential structural model errors introduced as a result of the merge feature. The logic is highly dependent on the definition of the model formalism, but in broad terms, it is intended to prevent the creation of models that would not otherwise be possible if the base editor alone were used. Since changesets are often highly interdependent, it is very easy to produce unintended be-

test_advise.adv

```
      <outcomes key="Outcome 1">
        <value name="Outcome 1" probability="return 1;" detection="return 0.1;" eff
      </outcomes>
      <timing>
        <parameters>return 2;</parameters>
      </timing>
    </value>
  </stepElements>
  <stepElements key="Step 1">
    <value name="Step 1" codename="Step1" xpos="539.0" ypos="183.0" attackExecution
      <outcomes key="Outcome 1">
        <value name="Outcome 1" probability="return 1;" detection="return 0.1;" eff
      </outcomes>
      <timing distributionType="Deterministic">
        <parameters>return 2;</parameters>
      </timing>
    </value>
  </stepElements>
  <stepElements key="Step 6">
    <value name="Step 6" codename="Step6" xpos="642.0" ypos="333.0" attackExecution
      <outcomes key="Outcome 1">
        <value name="Outcome 1" probability="return 0.5;" detection="return 0;" eff
      </outcomes>
      <timing>
        <parameters>return 2;</parameters>
      </timing>
    </value>
  </stepElements>
  <goalElements key="Goal 3">
    <value name="Goal 3" codename="Goal3" xpos="127.0" ypos="331.0" attackExecution
  </goalElements>
  <goalElements key="Goal 5">
    <value name="Goal 5" codename="Goal5" xpos="501.0" ypos="399.0" attackExecution
  </goalElements>
  <goalElements key="Goal 1">
    <value name="Goal 1" codename="Goal1" xpos="316.0" ypos="378.0" attackExecution
  </goalElements>
  <arcElements sourceName="Step 2" targetName="Goal 3" sourceType="Step" targetType
  <arcElements sourceName="Access 3" targetName="Step 1" targetType="Step"/>
  <arcElements sourceName="Step 1" targetName="Goal 5" sourceType="Step" targetType
  <arcElements sourceName="Step 6" targetName="Goal 5" sourceType="Step" targetType
  <arcElements sourceName="Knowledge 7" targetName="Step 6" sourceType="Knowledge"
  <arcElements sourceName="Access 2" targetName="Step 2" targetType="Step"/>
  <arcElements sourceName="Skill 1" targetName="Step 2" sourceType="Skill" targetTy
  <arcElements sourceName="Step 2" targetName="Goal 1" sourceType="Step" targetType
</aeg>
<adversary name="AdversaryName" codename="AdversaryName" costOfDetection=""/>
```

old_pc.tmp

```
  </stepElements>
  <stepElements key="Step 2">
    <value name="Step 2" codename="Step2" xpos="259.0" ypos="250.0" attackExecution
      <outcomes key="Outcome 1">
        <value name="Outcome 1" probability="return 1;" detection="return 0;" effec
      </outcomes>
      <timing>
        <parameters>return 1;</parameters>
      </timing>
    </value>
  </stepElements>
  <stepElements key="Step 1">
    <value name="Step 1" codename="Step1" xpos="539.0" ypos="183.0" attackExecution
      <outcomes key="Outcome 1">
        <value name="Outcome 1" probability="return 1;" detection="return 0;" effec
      </outcomes>
      <timing>
        <parameters>return 1;</parameters>
      </timing>
    </value>
  </stepElements>
  <stepElements key="Step 6">
    <value name="Step 6" codename="Step6" xpos="642.0" ypos="333.0" attackExecution
      <outcomes key="Outcome 1">
        <value name="Outcome 1" probability="return 1;" detection="return 0;" effec
      </outcomes>
      <timing>
        <parameters>return 1;</parameters>
      </timing>
    </value>
  </stepElements>
  <goalElements key="Goal 3">
    <value name="Goal 3" codename="Goal3" xpos="127.0" ypos="331.0" attackExecution
  </goalElements>
  <goalElements key="Goal 5">
    <value name="Goal 5" codename="Goal5" xpos="501.0" ypos="399.0" attackExecution
  </goalElements>
  <arcElements sourceName="Step 2" targetName="Goal 3" sourceType="Step" targetType
  <arcElements sourceName="Access 3" targetName="Step 1" targetType="Step"/>
  <arcElements sourceName="Knowledge 1" targetName="Step 2" sourceType="Knowledge"
  <arcElements sourceName="Step 1" targetName="Goal 5" sourceType="Step" targetType
  <arcElements sourceName="Step 6" targetName="Goal 5" sourceType="Step" targetType
  <arcElements sourceName="Knowledge 7" targetName="Step 6" sourceType="Knowledge"
  <arcElements sourceName="Access 2" targetName="Step 2" targetType="Step"/>
</aeg>
<adversary name="AdversaryName" codename="AdversaryName" costOfDetection=""/>
</models:AdviseAtomicModel>
```

Figure 3.5: Textual representation that corresponds to the differences shown in Figure 3.4.

havior or even broken models when one is hand-selecting individual changes to be discarded. Additional error-checking during that step alleviates the problem and also provides another level of insurance in the event that model source files are negatively affected, even during an automated merge or one that was handled outside of the Möbius graphical editor.

CHAPTER 4

IMPLEMENTATION DETAILS

4.1 Implementation Overview

As previously stated, we attempted to keep the implementation formalism-agnostic. We implemented the features described in Chapter 3 using the ADVISE formalism as a proof of concept. Since model formalisms can vary wildly in their structures, significant portions of our implementation are necessarily specific to ADVISE models, especially with respect to the graphical editor and its visual presentation of the model formalism. Discussion of the implementation details of the Möbius version control feature will thus be divided between the tangible code changes made to the ADVISE back-end and editor code, and the theoretical changes that would need to be made to support these features in another formalism.

Furthermore, we implemented many of the heavy algorithmic details involved with difference detection and classification via *EMF Compare*, an external open-source library. In this chapter, we will explore the relevant details to show roughly (in Section 4.2.1) how these concepts could be applied to other non-EMF-based implementations.

4.2 Comparison Pipeline

We define *Model comparison* as the task of generating a preview model from two similar models; it can be launched via the newly added "Compare..." contextual command, as mentioned in Section 3.1. The preview model should consist of one of the two original input models overlaid with a set of changes that, when applied, would transform it into the other input model. This structure, when presented to the user visually through the model editor, allows the user to identify differences between the two models more easily than would be possible using textual comparison.

Model comparison is split into three steps, as shown in Figure 4.1:

1. Differencing: Detect changes between the two compared model versions; this is handled by the *EMF Compare* library [13].

Figure 4.1: Model comparison pipeline.

2. Grouping: Group sets of low-level changes together based on affected model elements and type of change.

3. Model Construction: Build a temporary model for graphical display.

4.2.1 Comparison Algorithm

The *EMF Compare* library implements a similarity-based algorithm to solve the task of producing a list of differences between two models [21]. This implementation is *model-based*, i.e. the library operates directly on the model rather than on its textual representation. That technique better captures the higher-order semantics of interest and avoids *aliasing* problems that may be present in text. *Aliasing* refers to the possibility that multiple textual representations of a single model may exist because text is inherently ordered, but many model elements are not. For example, one can represent a model that contains two nodes in two different ways simply by swapping the order of the nodes' definitions in the text. Similarly, small changes can be made to a text file that do not affect the underlying model, such as changes to metadata headers and whitespace formatting. Instead, because the *EMF Compare* library acts on the model directly, these inconsequential details may be filtered out, decreasing the likelihood of false positives, i.e. detection of a change where none exists.

The algorithm is broken into two distinct steps: matching and differencing. First, the goal of the matching step is to find similarities between the two models being compared, to create a mapping from elements in one model to their counterparts in the other. That requires that the majority of elements in the models be identical (i.e. are unchanged), as each element's local neighborhood is a vital component of the determination of its similarity to any given element of the other model. That assumption generally holds for the Möbius implementation, as its rigid formalism structures offer at least a soft guarantee of similarity. For example, there is a finite and well-defined set of node types in each formalism, limiting the types of differences that may exist in a comparison operation. In cases where the models are not similar, the behavior of the algorithm implementing the matching step is still well-defined but is often of questionable use. Still, that scenario will generally occur only with

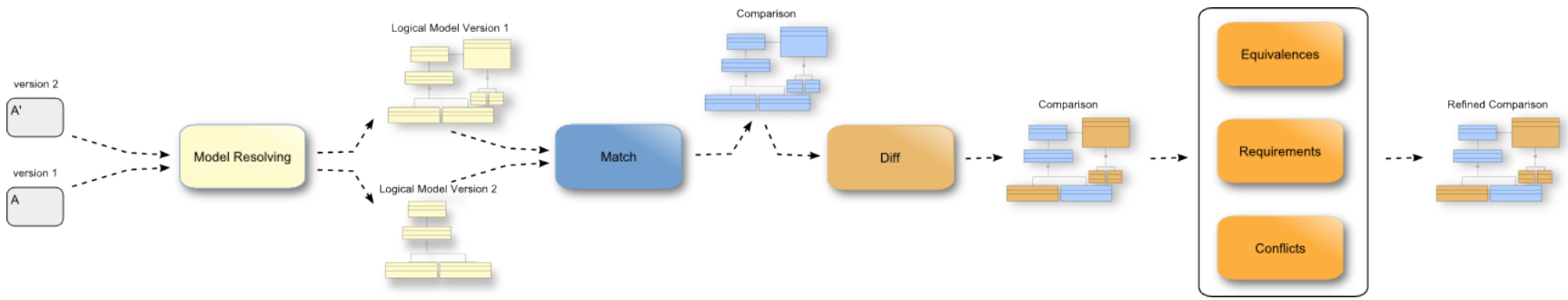

Figure 4.2: EMF comparison pipeline [20].

very large changesets, such as one reflecting a complete rewrite of a model. In such cases, a visual comparison would be of limited use since every element would be different. A higher-level description would be necessary to make meaningful assessments of these cases. Indeed, textual comparison tools also behave erratically when comparing two unrelated sources, and provide similarly poor comparisons.

Next, the differencing step compares element matches against each other, thereby identifying changes. Again, the reliability of this step depends on how similar the compared models are. One of the primary advantages of model-based development over conventional software development is that it allows developers to express systems using higher-level abstractions. Lower-order differences must then be aggregated into higher-order structures to be meaningfully displayed to the user. The differencing system built in *EMF Compare* is designed around the ability to drop in a replacement differencing engine, which was leveraged in the development of the VCS feature for Möbius.

Identified changes are categorized into three *types*, depending on which model elements they affect:

1. Reference: A new or deleted instance of an element, or a new connection between elements.

2. Attribute: A change in the member attributes of an element.

3. Resource Attachment: A specific case of the Reference type for changes at the root level of a model.

Furthermore, each change contains metadata that describe its state and relationships with other changes. One such metadatum stores the *kind* of difference that was identified, which can be one of the following:

1. Add: A new element in a vector field or a binding of a new resource attachment.

2. Delete: Identical to Add but identifying a removed element or unbinding of an existing resource attachment.

3. Change: A change to a scalar field.

4. Move: A reordering of values in a vector field, or the movement of an object instance between containers.

The *type* and *kind* of each change are vital to compiling the comparison results into meaningful high-level changes to be presented to the user (see Section 4.2.3 below).

The other metadata attribute that is important is the *dependency* attribute, which, as the name suggests, tracks the other changes that must be applied before the containing change may be applied. It is bidirectional (i.e. both the "requires" and "required by" relationships are stored). The *dependency* attribute is important both for applying or rejecting changes during the merge step, and for the grouping step mentioned above.

The matching and differencing steps are shown with the full pipeline in Figure 4.2. It is of particular interest that once differences have been found, a set of "requirements" is generated for each difference, representing dependencies (if any) between pairs of differences. That is relevant for the Möbius model comparison implementation because it uses that feature to determine whether a set of differences affecting a common model element should be grouped together.

For systems not based on EMF, it would be necessary to implement similar algorithms for the purposes of model comparison.

4.2.2 Möbius Comparison Pipeline

To begin a comparison operation, the current model on disk is fetched based on the containing project name, model type, and model name; that process is identical to the standard open operation used when a model is being edited. A previous version of the model is also fetched from the Git version-controlled repository and written as a temporary file in the project model directory. The file containing the previous version is then loaded into memory in the same way that the current version was. Before proceeding, we check the model type to verify that it supports graphical comparison; if it does not, the operation is aborted, and control is passed to an external textual comparison tool. That feature allows us to support both nongraphical models (or project components that are not models) and model formalisms that do not yet support graphical comparison. A simple function is added to the common base class of each formalism, so that the function can be switched on easily for future development.

EMF uses a system of XML-based "resource sets" for data storage, allowing for easy conversion between in-memory models and on-disk XML files. Möbius models are generally nicely contained within single files, which trivializes the model resolution step of the *EMF Compare* library. The loaded models are marshalled into formats expected by *EMF Compare* and sent through the pipeline, resulting in a Comparison object from which a list of *diff* objects are derived, each containing a low-level change that has been identified between the two target models at the Java implementation layer of each formalism. For convenience for the remainder of this thesis, the term "diff" will be used to refer to these objects produced

by the *EMF Compare* algorithm.

4.2.3 Differencing and Grouping

The raw differencing data produced by the *EMF Compare* library are not initially in a state fit for user consumption. Unfortunately, the diffs provided by the *EMF Compare* library are not sufficient to provide a useful high-level presentation of the model comparison as the changes are too fine-grained to be practical. Möbius model formalisms may contain a significant amount of internal complexity, which is needed in order to abstract away low-level details of the underlying discrete-event models. Thus, a single change in the graphical editor will often translate into multiple changes in the back-end. For example, in the ADVISE implementation, the addition of a Goal node results in three identified diffs, listed below with the change *type* and *kind* (previously described in Section 4.2):

1. Reference Change: Update ADVISE AEG.
2. Reference Add: Update name string to node mapping for new Goal node.
3. Reference Add: Instantiate new Goal node.

As the number of detected changes grows, this list of diffs quickly becomes overwhelming if presented directly to the user. Without explicit knowledge of internal implementation details of Möbius, it is also not at all clear how the individual diffs are related to one another, and those internal details are obviously beyond the scope of knowledge that can reasonably be expected of a typical user. Just as model formalisms abstract away low-level details for the sake of convenience and ease of development, so too should the presentation of the newly added comparison feature. To that end, all of the detected diffs need to be grouped, such that they are organized into *diff sets* that consist of related diffs that describe a single high-level change. In the above example, the high-level change is the addition of the Goal node, which corresponds to the *diff set* containing the three listed diffs.

During the grouping stage, the sets are also sorted and grouped by the *kind* of each change. While each diff is tagged with a *kind*, those tags describe only the diffs to which they belong. *Diff sets* are composed of multiple diff instances, and there is no guarantee that they share a single *kind*. In the above example, a high-level *add* operation (i.e. adding of a new node to the model) generated a combination of *add* and *change* diffs. For the purpose of generating meaningful graphics for the user, each *diff set* must be assigned a single *kind* based on its constituent diffs. Granted, a part of this classification step is just semantics. For instance, if a node attribute is changed from the default value to a user-defined custom value, it could

be argued that such a modification is either a *change* (the value already existed and is now different) or an *add* (the value was changed from the default to something more meaningful). In such cases, the behavior depends entirely on the formalism implementation. The specifics of such a discussion are beyond the scope of this thesis and will need to be handled on a case-by-case basis if it becomes a serious consideration for the ADVISE implementation or any future implementations of other formalisms.

The *EMF Compare* library produces a simple array of diff objects, with no guarantee of relative ordering. Luckily, diffs that belong together as a single high-level group are generally easy to detect because of their interdependencies. If we consider the ADVISE Goal node example discussed earlier, the first reference change (update ADVISE AEG) is dependent on the second change (update string to Goal node mapping), which is in turn dependent on the third change (Goal node instantiation). Since dependencies are stored for both directions, it is simple to traverse the linked structure to collect the remaining members of a group after one is found. However, in practice, most changesets will be far more complex than just the addition of a single node.

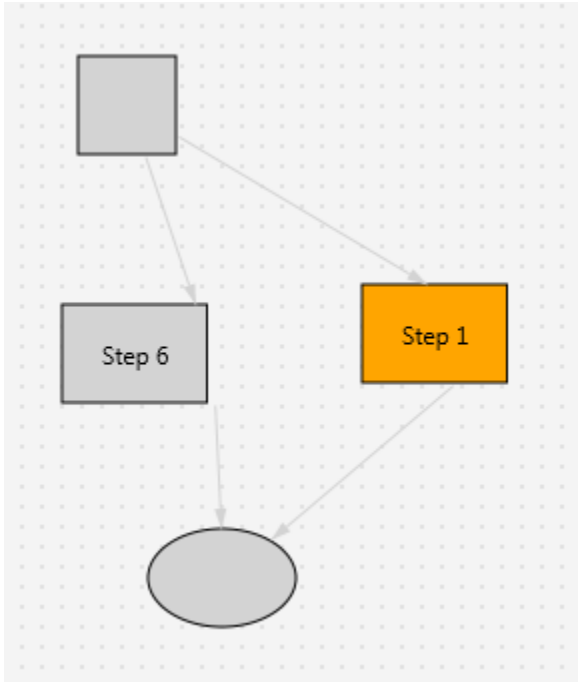

Figure 4.3: Visual comparison display after Step 1 has been added and its distribution parameter has been modified.

Figure 4.3 shows a slightly more involved example. Here, a new Step node has been added with two associated arcs, one from a preexisting Access node to the new Step node, and another from the Step node to a preexisting Goal node. In addition, the *timing distribution* of

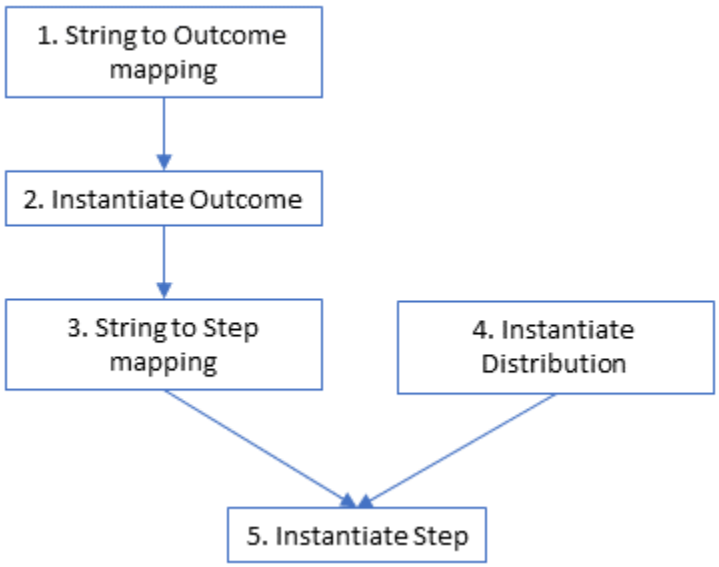

Figure 4.4: Associated diffs found by *EMF Compare* while processing the changeset shown in Figure 4.3. Arrows indicate a "depends on" relationship.

the Step has been adjusted from its default. Step nodes drive runtime changes in the ADVISE formalism [8], and very few of the nodes' constituent properties are shown graphically in the AEG, much as activity nodes' attributes are not displayed graphically in the SAN formalism. In particular, each step is governed by a *timing distribution* that describes when and how often the Step triggers, as well as one or more *outcomes* and matching *code effects* that effect change when the step is triggered. Since these properties are not graphically visible, the changeset described in this example is significantly more complex than the addition of a Goal node despite the two changesets' similar appearances in the AEG editor. That complexity is reflected in the set of generated diffs:

1. Reference Move: String to outcome mapping.
2. Reference Add: Instantiate Outcome associated with new Step.
3. Reference Add: Update string to node mapping for new Step node.
4. Reference Add: Instantiate new Distribution property for Step.
5. Reference Add: Instantiate new Step node.

For the sake of clarity, diffs related to the AEG arcs are not shown in the above list. The structure of interdependencies among the listed diffs is shown in Figure 4.4. In the abstract,

the two changes identified here are the addition of a new Step node and the modification of its *timing distribution* property. Both require that there first be an instantiated node object present in the model, indicated by element #5 in the figure. The left branch of the tree corresponds to the steps involved in adding references and making the added Step node known to the rest of the model, while the right branch corresponds to the adjustment of the Step node's *timing distribution* property. In this case, the two branches are treated as separate changes (i.e. the user can view and handle them independently) both for clarity during a potential merge and for greater granularity during that merge. In addition, this treatment is consistent with how node properties are managed in general. However, there is a significant downside of this treatment, in that the two high-level changes exhibit a one-way dependency that is not reflected in the figure. To be specific, modification of the *timing distribution* requires that the node be added, but the reverse dependency is not true. that will be addressed in Section 4.4.2.

Algorithm 1 shows the full pseudocode implemented as part of this thesis work for the diff grouping stage. The conditional on line 7 checks whether each diff is "root-level" (i.e. a diff not required by any other diff), and the subsequent loop classifies each root-level diff into one of three output arrays depending on its *kind*. As part of this classification procedure, the dependency tree of each root-level diff is traversed to determine the diff's *dependent group*, which contains all diffs that must be applied to the model before the root-level diff can be applied. For quick access, those dependent groups are stored as a mapping from root-level diffs to their corresponding *dependent groups* on line 5.

To determine the *kind* of each group, it is not enough just to look at the *kind* of the root-level diff, as it is often marked as a *change* or *move* because of the structure of the formalism implementation. Instead, the *kind* of the group is determined by the presence of either an *add* or a *delete*. Broadly speaking, no single change in the model can result in both an addition and a deletion. A modification to an existing element may be seen as such, but it would always be detected in *EMF Compare* as a *change* instead. The converse of that property is not true (i.e. a change may result in the detection of an *add* diff), but such cases depend on the definition and semantics of an *add* or *delete*, as previously mentioned. Therefore, this algorithm assumes that the presence of any diff labeled as an *add* or *delete* determines the *kind* of the entire group. If neither an *add* nor *delete* is found, the group is classified as a *change* on line 27 by elimination, as there is no high-level difference between a *change* and a *move*.

Because root-level diffs are defined as diffs that are not required by any other diffs, it is possible to have diffs that appear in multiple dependent groups, such as the bottommost diff in the example in Figure 4.3. That has the benefit that each dependent group is self-

Algorithm 1 Diff Grouping Algorithm.

Input: *diffs*: List of *EMF Compare*-detected diffs
Output: *newDiffs*: List of root-level diffs that correspond to added elements
Output: *deleteDiffs*: List of root-level diffs that correspond to deleted elements
Output: *changeDiffs*: List of root-level diffs that correspond to changed elements
Output: *diffGroups*: Mapping of root-level diffs to their corresponding dependent groups

```
function GROUP(diffs)
    newDiffs ← empty array
    deleteDiffs ← empty array
    changeDiffs ← empty array
    diffGroups ← empty dictionary
    for all D in diffs do
        if D.RequiredBy() is empty then
            traversalStack ← empty array
            traversalStack.push(D)
            group ← empty array
            typeSaved ← False
            while traversalStack is not empty do
                curDiff ← traversalStack.pop()
                groups.push(curDiff)
                if typeSaved = False then
                    kind ← curDiff.kind
                    if kind = ADD then
                        newDiffs.push(D)
                        typeSaved ← True
                    else if kind = DELETE then
                        deleteDiffs.push(D)
                        typeSaved ← True
                for all R in curDiff.Requires() do
                    traversalStack.push(R)
            diffGroups.put(D, group)
            if typeSaved = False then
                changeDiffs.push(D)
```

contained and that one can create a syntactically correct model by applying any individual dependent group without considering the others. The downside is that syntactical correctness (i.e. can be compiled and displayed in a graphical editor) does not necessarily imply logical correctness (i.e. the model makes sense from a high-level perspective), so a model produced from the above method may still produce errors when analyzed. For example, while arcs between nodes appear to be dependent on the existence of those connected nodes, they in fact are not in the underlying AEG implementation. As a result, diffs that affect arcs will not identify those nodes as dependencies and separate dependent groups will be formed for the diffs affecting an arc and its connected node or nodes. Thus, there exists an asymmetrical dependency between the those separate groups and, consequently, a risk of producing an illegal model, as in the example shown in Figure 4.3, but more difficult to address, since the groups are more isolated. This problem will be discussed in Section 4.4.2.

4.2.4 Preview Model Construction

After the diff *dependent groups* have been produced and their associated *kinds* have been determined, a preview model must be constructed, to be viewed in the Möbius graphical editor. An in-memory model that represents the previous version will have already been created, as detailed in Section 4.2.2. One can then easily generate the desired preview model by applying all of the detected diffs on top of the previous model. Care must be taken, however, since the changes must satisfy the following properties:

1. Deleted items (i.e. items present in the previous model but not the current one) must be temporarily added to the preview model in order to be displayed in the editor.

2. Changes must preserve their *kind* data to enable visual identification of how the item was affected between the two models: added, deleted, or changed.

3. All changes must be individually reversible to allow arbitrary discarding of changes.

4. The final model must be legal regardless of which changes are discarded.

Items #2 and #4 in the above list will be discussed in Sections 4.3 and 4.4.2, respectively. The remaining two items are handled during model construction. Regarding item #1, since the resulting model uses a Möbius graphical editor for visual presentation, any element that is to be displayed must exist in the model. This means that deleted changes cannot be applied to the previous model, as that would remove the corresponding elements. Instead, they are catalogued with the set of commands that *would* be taken if the corresponding

model elements were removed, and the elements are instead **marked** as deleted for visual identification in the editor. Special cases for processing these diffs are also included later, when merge operations are being handled.

Item #3 is present in this list to support the merge feature (see Section 4.4.1), which allows the user to hand-select which, if any, diffs they would like to include in the merged model. To that end, each diff output from the grouping stage is mapped to a command that, when applied to the preview model, produces the model that would result if the diff were accepted and applied. Luckily, EMF already supports that feature in its Command object. Originally intended to support the ability to undo and redo changes in a model editor, Command objects can be applied in both directions (changing the model, or reverting a previously made change). That same Command objects can also be used to automate model changes since they are supported natively by *EMF Compare*, allowing diff objects to be translated to their corresponding Commands via the default merger. Using that translation process, a Command is generated for each diff and stored as a mapping from diff groups (represented by each group's root-level diff) to Commands; that makes it possible to select Commands to be recalled later and reverted. In order to allow reversion of an arbitrary Command, the model maintains an ordered list, which contains all of the Commands used to generate the preview model from the previous, starting model. As discussed above, only Commands associated with *delete* diffs are applied to the model at this stage of the pipeline, and they are implemented through overriding of the default *EMF Compare* merge handler by one that simply marks the affected node elements that are to be visually highlighted in the editor instead of deleting them.

When that process is complete, the preview model has been constructed; it is identical to the current on-disk version except for the presence of the deleted elements. All changed elements are also marked so that the associated changes can be displayed. All relevant data generated by this stage of the comparison operation are passed, along with the model, to the graphical editor for presentation to the user.

4.3 Graphical Changes

The full graphical pipeline is entangled with GEF and will not be described here since the details are not within the scope of this thesis work. Instead, this section will focus on the modifications made to the normal editor pipeline to allow for graphical comparison. It consists of three sections, which discuss changes made at the shared project component level, changes made in the editor pipeline, and ADVISE-specific changes. That categorization may

be helpful to readers who are seeking guidance when adding support for future formalisms (see Appendix A).

4.3.1 Project Component Changes

The vast majority of the heavy lifting involved in graphical presentation and UI handling is built on top of GEF and may be shared between the implementations of multiple Möbius formalisms and their editors. As much of the VCS feature as possible should be implemented in that shared infrastructure layer, as only the visual presentation may be considered formalism-dependent. Most formalisms can be expressed as mathematical graphs with only minor exceptions when a unique feature is present in a particular formalism. For example, cases on an activity node in a SAN model are visually displayed as part of the node in the graphical editor (see Figure 4.5), which does not align with the traditional definition of a graph. It may be difficult to support the VCS feature on top of formalisms containing nodes that define such significant visual differences (e.g. beyond simple color or bounded shape changes) that are dependent on their constituent parameters. Formalism-specific code will be needed to handle such peculiarities, depending on the desired presentation when displaying comparison results. For instance, if a new case is added to an activity, should the individual case be highlighted as *new* while the activity and other cases are ignored or should the entire activity be highlighted as *changed*? Of course, the answer is highly dependent on implementation details. Development efforts for this project focused on keeping the shared code as general as possible to potentially support either choice in that example.

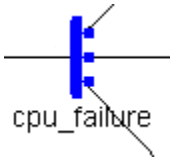

Figure 4.5: An Activity node in a SAN model; this activity has three cases, which are represented by the circles on the right.

Möbius, like a traditional software development IDE, maintains and manages self-contained *projects*, that contain all of the models, parameters, and solvers needed for end-to-end development and simulation of a formal system model. Such models and solvers are referred to as *project components* and form the root class from which any formalism is eventually derived. The *project components* are what is passed to the editor when a model file is being opened for visual presentation and they must contain all of the data needed to generate the graphics within the editor and, during the prior comparison stage, the *project components*

are updated with the grouped differencing data. To support this data storage, a number of changes at the project component level were needed:

1. An identifier must be added to indicate whether or not the component type supports graphical comparison.

2. An identifier must be added to indicate whether or not the component is **currently** opened as a preview model undergoing comparison.

3. Interfaces must be added for accessing the grouped diffs and their associated commands.

4. A method must be implemented to internally mark changed elements based on the contents of input diffs.

The first two items are present to ensure that existing features remain functional alongside the new features. Project components that are not graphical (e.g. textual models or non-model components) or do not yet have full implementations to support the VCS feature are redirected to a textual comparison when a user attempts a comparison operation. Those two items are implemented as inheritable boolean-returning functions for indicating status, with default values of "false." As new formalisms are supported, they can override those functions to enable the VCS feature. Furthermore, significant rewiring of some editor features was necessary to enable live model comparisons, and these changes should not propagate back to the normal editor. To prevent such propagation, preview models must identify themselves so that the editor can be initialized in the new "diff" mode.

Items #3 and #4 represent the core functionality of the visual comparison feature and complete the back-end tasks that must be performed before the model is displayed in the editor. The interfaces for supporting those requirements are defined at the project component level, as is the allocation of memory for the data specified in requirement #3. The current implementation of the ADVISE formalism is unfortunately still unique, so the functionality required in item #4 must be added at the ADVISE level; it will be discussed in more detail in Sections 4.3.2 and 4.4.2. In theory, it may be possible to move some of the needed changes closer to the root project component class, but refactoring of the code is likely to be a significant undertaking and is beyond the scope of this thesis.

4.3.2 ADVISE Editor Changes

Model elements are represented graphically in the editor and are defined by formalism-specific implementations for translating element properties to their appropriate visual rep-

resentations. For example, the type of an ADVISE node (e.g. Knowledge, Skill, or Step) may be considered a property of the node and is represented by different shapes in the editor, depending on the type; however, the *timing distribution* property of a Step node is not visually represented, and that distinction must be made in the formalism implementation. There does not currently exist a set of shared base classes that implement skeleton versions of common graph elements, like nodes and edges. Since only colors are changed when visual elements are marked during a comparison, the changes should be made at the hypothetical shared base level. As significant work is still needed to transition other formalisms to the new EMF+GEF framework, that layer has not yet been implemented. Code changes for updating the element colors have thus been implemented as a part of ADVISE-specific code. However, those changes could be moved out after a codebase refactoring in the future.

As previously mentioned, an instance of a project component is passed to the graphical editor to initiate the GEF graphical pipeline that ends with the desired visual model. Changes were needed here as well to parse the newly added data in the project component under comparison:

1. A pull-out panel UI must be implemented that contains an enumeration of all diffs currently under consideration along with a supporting interface for merging individual diffs. The UI will supplement the changes to the graphical model, and will detail each change with a short text description and provide the user with a single summary of all unresolved changes between the two models under comparison. The UI will also provide a way to direct user attention to changes that are not rendered graphically as part of the model, such as those affecting node attributes.

2. A method must be implemented to force a redrawing of a model's graphics if the model is internally updated, in order to maintain coherence between the back-end model and front-end visual representation of that model.

3. Handler behavior must be modified so that a "save" command redirects the preview model to its on-disk counterpart, reloads the in-memory version within Möbius, and performs a sanity check for unresolved changes. The main objectives of these changes are to enhance usability, to allow a model developer to make changes to a model under comparison, and to propagate changes seamlessly to other parts of the Möbius project.

4. Visual elements must be modified to support displaying different colors based on project component markings when the editor is open in "diff mode." That is the primary objective of the model comparison feature and draws attention to the specific model

elements that are different in the two models under comparison, while deemphasizing unchanged elements.

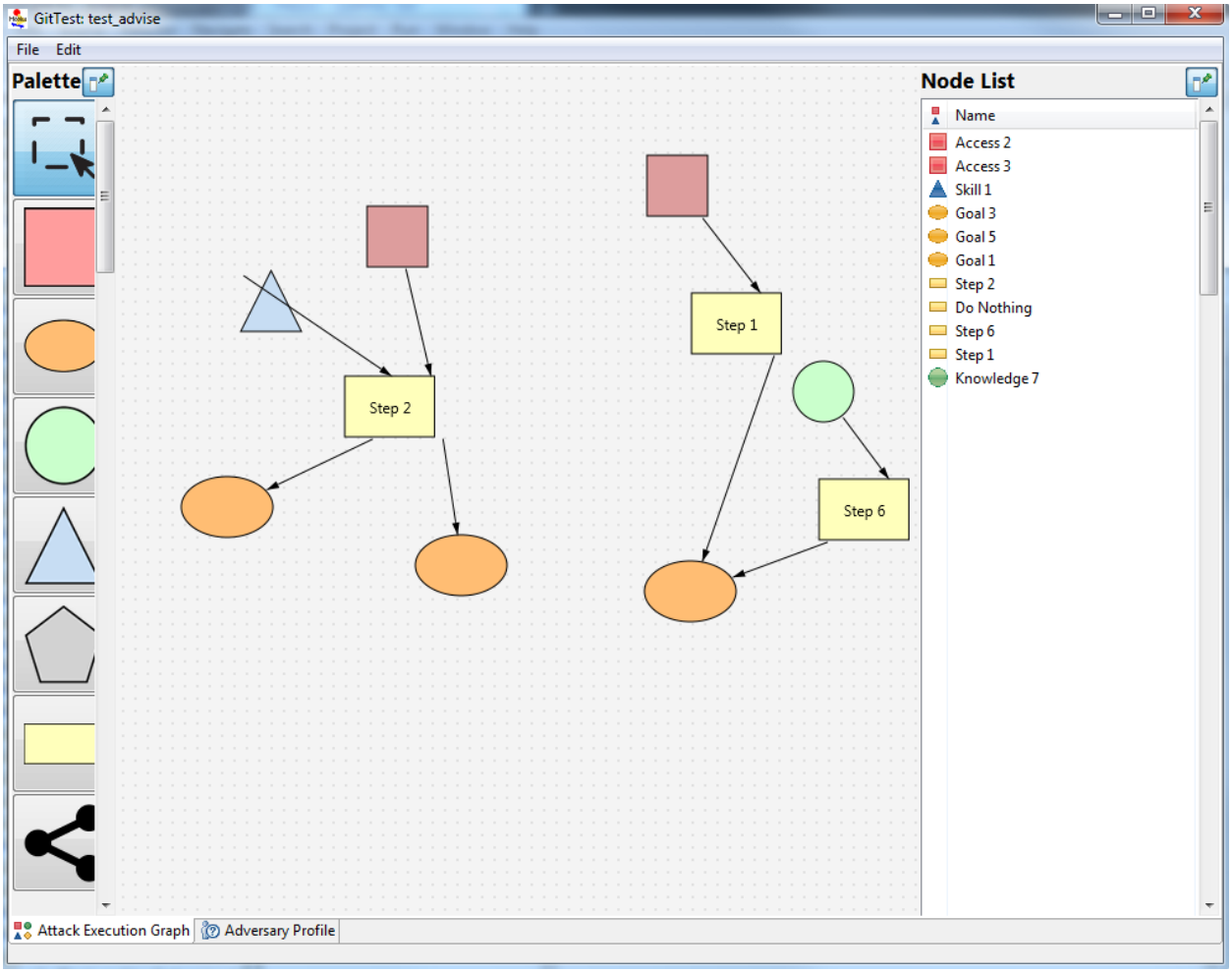

Figure 4.6: ADVISE graphical editor under normal use, with the palette and node list panels visible.

Diff enumeration panel

The ADVISE graphical editor currently supports two pull-out panels, with one on either side of the main editor window, that contain the parts selection menu on the left and a node list on the right, as shown in Figure 4.6. While the parts menu is still important for making fine-tuned changes to a model under comparison, the node list is far less useful, as the focus of a comparison operation is on the small changes between versions rather than the overall list of nodes. As such, the node list panel is replaced with a diff list when the editor is run in "diff mode". Just as an element is

selected in the main editor window when its corresponding node is clicked on the side panel, clicking on a diff in the panel while the editor is in "diff mode" will select the corresponding affected node in the main window. That poses a problem when diffs that affect arcs are selected, as arcs are normally not selectable. That is by design as arcs have no nonvisual properties and have very restrictive placement rules (i.e. they must be connected to a single node at each end). The original intentions behind the design of unselectable arcs still hold in "diff mode" so it is undesirable alter that behavior. Instead, the source node is selected instead of the arc itself. The overall purpose of the visual presentation is to draw user attention to the affected elements, and selecting a connected node rather than the arc still fulfills that objective. Support for merge options is described in detail in Section 4.4.1.

Drawing refresh

During normal use, changes to models can occur only in the graphical editor; changes are made by the user, and the editor propagates the changes to the underlying model upon receipt of a "save" command. However, during a comparison operation, changes may be made to the model via the merge support feature. If that happens, the changes are applied to the model directly, circumventing the editor. The editor does not normally poll the underlying model for changes, as there was no need to do so in the normal case. To support the merge feature, it was necessary to add a mechanism by which changes can be propagated in the other direction (i.e. from the model to the editor) to ensure that the visual presentation available to the user is kept synchronized with the actual model. Without that feature, a live merge operation within the editor would be clunky and unintuitive, as there would be no way to update the current preview of the model.

Save handler

Open models in Möbius normally retain paths that point back to the models' on-disk source files. Since a preview model is generated entirely in memory, bypassing the normal loading process, it lacks that reference to the corresponding source file. While the source path is unnecessary for the creation of the preview model, a user cannot save edits made to the preview model without the path reference. As described in the previous section, Möbius stores a copy of each project component in memory after the components have been loaded from disk during Möbius's startup sequence. Those copies are not expected to change during the lifetime of the application unless they are edited directly in the editor by the user. Again, changes made in the preview model do not affect those copies and changes must be propagated to the Möbius application

to ensure coherency between the various versions of the affected model. We resolved this problem by forcing the save handler to reload the component from disk if the editor is opened in diff mode. Finally, to prevent the user from unintentionally saving changesets, the handler outright rejects preview models that still have active unresolved (i.e. not merged or discarded) diffs.

Visual elements

Before the preview model is loaded in the editor, a method is run to internally mark all elements as "unchanged." Then, all elements that are affected by the set of input diffs are marked appropriately based on the *kind* of the associated diff. These markings are arbitrary and hold no intrinsic meaning. In consequence, the graphical editor must be updated to interpret the markings when rendering the elements. To preserve the normal appearance of the model, only colors are changed to reflect their status: unchanged elements are grayed out, new elements are green, deleted ones are red, and changed elements are orange. When the editor is opened for normal use, models simply mark each element as "normal" instead of "unchanged" during initialization, causing the editor to default to the elements' normal coloring schemes.

4.4 Merge Support

While a visual comparison is useful, it does not provide the user with any additional tools to make changes to the model beyond what was previously available. Making manual changes to a model through the editor is flexible, but fragile for the typical use case of combing through comparison results and making detailed changes. In the usual software development pipeline, dedicated source commits are often made in order to store snapshots before and after a compare and merge operation. Indeed, merging is an easy way to introduce errors and other unintended behavior, as the merged overlapping portions of the two compared versions may no longer be consistent with the remaining nonoverlapping parts after the operation. The problem is further exacerbated if users are expected to make such edits by hand, manually copying or deleting changes as they deem necessary.

4.4.1 Merge Pipeline

In most cases, the overwhelming majority of work in a merge operation consists of simply accepting (keeping) or rejecting (discarding) each change detected between the two versions. In this context, *accept* means to carry a change forward into the final model and *reject* refers

to reverting a change, restoring the element or value that was present in the original version. To that end, a simple interface is provided to the user in the model editor, allowing him or her to make decisions at the granularity of grouped changes rather than at the level of the low-level differences identified by the raw comparison algorithm (see Figures 4.7 and 4.8). Of course, manual changes are sometimes necessary at the merge step if small changes are needed to maintain behavioral consistency of the model, so normal editor features are still available to facilitate that occasional use case.

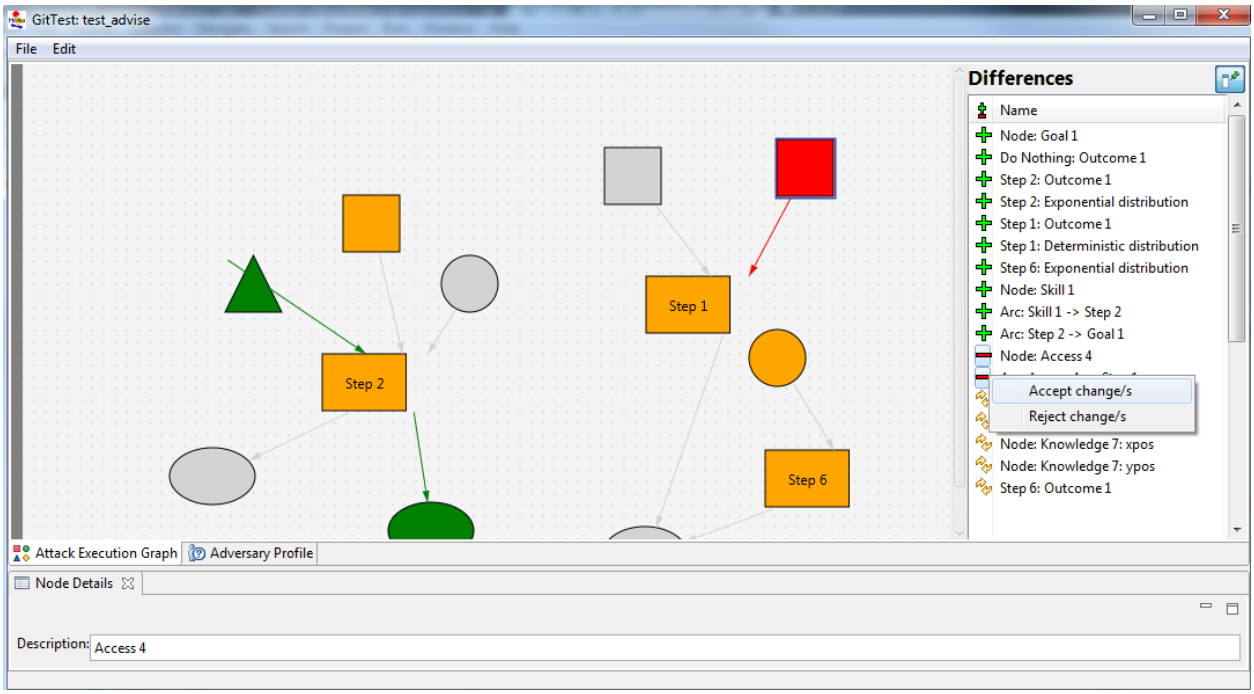

Figure 4.7: ADVISE AEG editor showing merge options.

In order to allow individual changes to be thrown out while others are preserved, the grouped differences found during the comparison stage are preserved along with commands that contain reversible instructions on how they affect the model when applied. Because the model displayed in the editor is a fully featured model (except that it is stored in memory with no on-disk counterpart), any pending changes presented to the user have actually already been merged into the underlying model. Care must be taken to ensure that undesired changes made to the model for the sole purpose of visualization are removed when the model is being saved to disk, as the changes are not representative of the state of the real model, only that of the preview model. To that end, the save handler for the diff mode editor checks to make sure that all outstanding diffs have either been accepted or rejected before it allows the model to be saved, as described in Section 4.3.2. That prevents deleted elements from being unintentionally saved into the model.

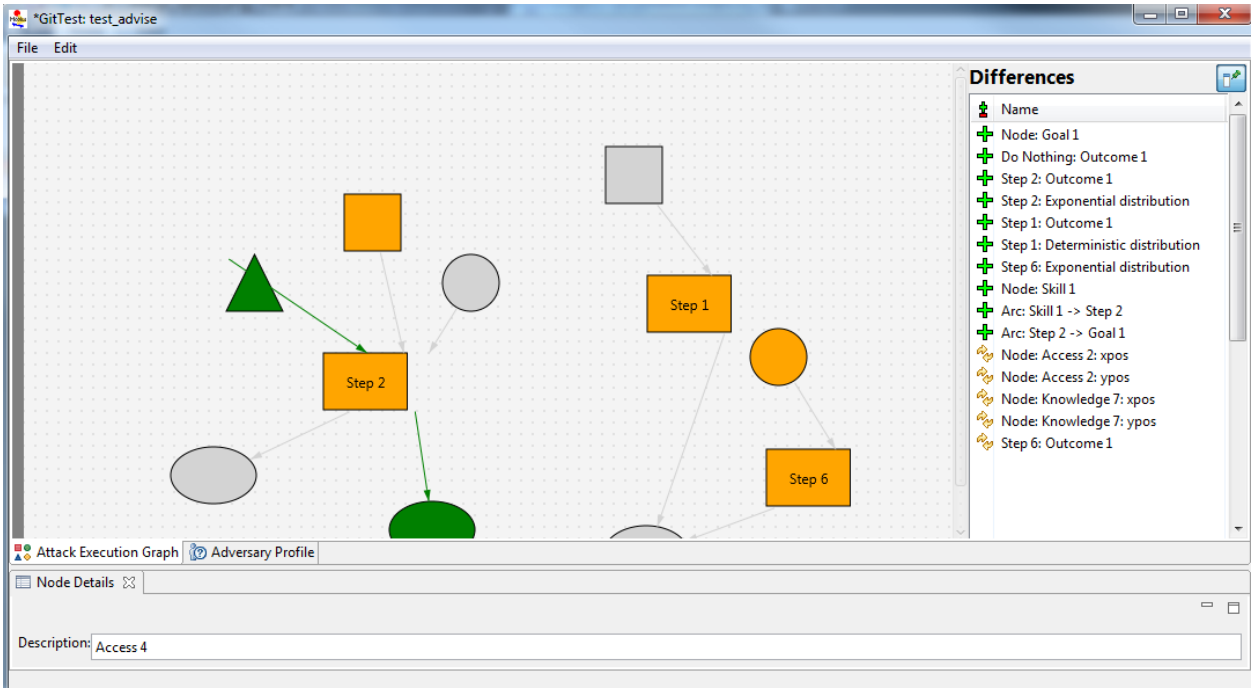

Figure 4.8: Editor after accepted deletion change.

When a diff is selected to be accepted, it is marked as such and removed from the pending list shown in the editor panel. No other work is necessary, since the presence of those associated elements means that the diff has already been applied to the model. However, the reverse operation is more involved. When a diff is rejected, the associated command must be reverted and the model must be rebuilt from the remaining diffs. Either way, the model in the editor must be redrawn, either to gray out elements that belong to accepted diffs or to remove elements associated with rejected ones.

First, a new model must be built in which the commands associated with the rejected diffs are omitted. Because of the data stored within each command, it is not possible to revert arbitrary commands. Each command stores a precise definition of the change that would be needed to apply and revert the command, down to the granularity of, for example, adding an element to an array at a particular index. If an element is added into the same array by a command executed after the reverted one, the index might now be incorrect (e.g. out of bounds or pointing to the wrong value) and might result in errors during model construction. However, the stored definition for each command is generated when the command is applied and not when the command is first generated. Thus, as long as the model is torn down in the reverse of the order in which it was originally built, commands can be omitted without problems. Therefore, the model stores the commands that were applied to the preview model as a stack, in order of application. When a command or set of commands is to be reverted,

the stack is emptied, one command at a time, until all of the selected commands have been removed. As each command is popped off the stack, its change to the model is reverted and the command is set aside. Then, each of the temporarily reverted commands are reapplied to rebuild the model without the reverted commands. It is worth noting here that the reapplied commands may be applied in any order; they do not need to be reapplied in the original order.

Next, the visual model must be refreshed. It is insufficient simply to mark the affected model elements based on the set of accepted or rejected diffs, since there may be other unresolved diffs that affect the same elements. Instead, to accomplish the refresh, all model elements are marked as unchanged, and the remaining set of unresolved changes is used to mark each element appropriately, following the same procedure that was originally used during initialization.

A significant amount of repeated work is done as part of the merge operation in order to cover all edge cases. Performance was not a major concern during the design of this system, as diff sizes were assumed to be small. Indeed, performance concerns with the ADVISE implementation appear to be negligible for models containing dozens of nodes, with no discernible delay observed during merge operations.

4.4.2 Model Validation

It was never intended that graphical Möbius models would be edited manually outside of the provided editors. Models can easily be broken both logically (made invalid due to an inconsistency with respect to the formalism definition) and behaviorally (left logically valid but unable to evaluate or simulate as expected). Before the introduction of the merge feature, the only risk of changes being introduced outside of Möbius editors was if the user knowingly made them in an external text editor. Now, with the merge feature, it is possible to produce logical or behavioral errors within the editor.

Broadly speaking, models can be made invalid through a merge operation in two ways: via disconnected arcs and because of attributes that are attached to nonexistent nodes. The small number of possible error sources is due to the limited set of diffs that may be returned by the model comparison algorithm, assuming that both inputs to that algorithm are valid.

In the case of disconnected arcs, if a node and attached arc are added between the two versions of a model, a minimum of two diffs will be generated by the comparison algorithm: one associated with the node, and another with the arc. The reason is that there is no inherent dependency between an arc and its attached nodes in the internal implementation of the node and arc model elements; the only system in place to prevent a disconnected arc

is the UI of the graphical editor, which, as established above, is circumvented during a merge operation. Furthermore, these diffs should not be presented together since the dependency here only flows in one direction. Specifically, the arc can be rejected and removed from the model, leaving a disconnected node. While that may not have been the intention of the model developer, the result is a perfectly valid model. However, the reverse is not true: rejecting the node but leaving an arc disconnected on one side is not allowed and should be prevented.

Unattached attributes are the other way by which errors may be introduced. As previously discussed, each node contains one or more attributes, and changes to these attributes appear as individual diffs during model comparison. Since a node may contain more than one attribute, these changes cannot be combined, since the user may want to accept some but not others. However, none of the attributes hold any meaning if the associated node is not present in the model.

To account for those asymmetrical dependencies while maintaining model validity, it is necessary to treat changes to nodes differently when the nodes are removed from the model as the result of a merge. When a set of diffs is selected for resolution (i.e. accept or reject), the set of diffs is searched for any that may remove a node. If one is found, all remaining diffs are searched, and any associated with attributes that belong to or arcs connected to the diff in question are removed along with that diff. Thus, a user who selects only a single diff to be rejected in the UI may see multiple diffs removed when that action is resolved. Of course, if a node is added to the model, none of those steps are necessary, since all of the remaining diffs related to the added node can be resolved either way without affecting the validity of the model.

CHAPTER 5

CONCLUSION AND FUTURE WORK

In this thesis, we discussed the problem of source control by drawing parallels with modern software development practices and the problems addressed in that field by the introduction of source control, emphasizing increased productivity in collaborative environments and improved debugging potential utilizing a project's history. We briefly discussed the supporting bodies of work that were brought together in this thesis work, including Möbius, the ADVISE formalism, graph comparison, and visual representation of graph comparison. We then highlighted the features we added to Möbius and presented a working implementation consisting of native integration of the Git VCS, support for visual comparison of graphical models, and a user interface for convenient resolution of detected changes between compared models. Finally, we discussed the complexities involved in model validation introduced as a result of the above changes and the manner in which they were resolved.

5.1 Future Work

While the current implementation of the VCS feature in the ADVISE formalism is functional and provides new tools for Möbius model developers, it is still lacking in a number of areas.

5.1.1 Extension to Other Formalisms

ADVISE is relatively young, and most end users are working with older, more established formalisms. Further development efforts will be needed to add support for other formalisms in the future. Because Möbius supports a great variety of formalism definitions, each formalism will require a unique implementation to integrate VCS features. However, the work done for this thesis provides both a shared core layer that implements the fundamental comparison functions needed to generate a list of changes, and a standard interface for accessing those data in any model formalism built on EMF. The majority of future work will address support for visualization and model validation for each formalism.

5.1.2 Core VCS Integration Improvements

The current implementation strictly follows the Git system for version control and already includes a Git-specific idea in the separate "commit" and "push" commands. Although Git is a popular VCS, it is far from being the only one, and support for other systems would be welcomed by users working in environments without Git ecosystems. Furthermore, the current core VCS integration provides a limited set of operations that cover only some of the most commonly used commands. Support for other commands used in common workflows (e.g. branching, tagging, and rebase/merge/cherry-pick) would be useful.

While these commands would be useful with a simple command line interface in Möbius, they would be more powerful with tighter integration into the Möbius tool. For example, the currently active branch or tag would be better displayed as a persistent part of the project interface rather than fetched on command by the user, especially for workflows utilizing development branches and tagged releases. Commands related to the version history of a project, such as the *rebase* and *cherry-pick* commands mentioned above, would be better represented visually, similar to the *gitk* utility that allows a user to view the revision graph of a project with text previews of each version. Such a tool could be implemented natively in Möbius, providing previews of the graphical models instead of their textual representations.

5.1.3 Diff Editor Interface Improvements

A number of improvements could be made to the interface that is presented to the user after a model comparison:

1. Merge operations are currently implemented as permanent changes to the preview model with no rollback mechanism. If a user mistakenly accepts an unwanted change, he or she will need to discard the entire merge operation and start over. Implementation of an undo/redo system would remove that inconvenience. It would be best if that support were provided in the native graphical model editor.

2. Currently, the only way to select a diff for resolution is to find it in the diff enumeration panel. To take better advantage of the GUI, it may be useful also to allow all diffs associated with a node to be selected when the node is selected in the editor. That would require additional data structures to store the mapping from nodes to sets of diffs and mechanisms for changing and displaying the current selection without direct user interaction with the diff enumeration panel.

As with many UI changes, many of those improvements would likely be user-driven based

on common use cases, and would consequently require significant user feedback to determine which improvements it would be useful to implement.

5.1.4 Three-way Diff

In addition to comparing past and current versions of a file, another common use case for source comparison occurs when one is attempting to resolve differences between two separate local copies of a model and the last most up-to-date version. Of course, one could handle that situation simply by performing two normal compare or merge operations, but that would not be ideal, as users would be left to their own devices when tracking elements affected by changes in all three models. Such a *three-way diff* is needed most often in collaborative environments, especially those that follow a "development branch" style of workflow. The *EMF Compare* library supports three-way diff, but additional work will be needed to extend the existing diff grouping algorithms. In particular, the *kind* label assigned to each group will need to be enhanced with additional data to identify the relationship between the diff group and the two possible source models. For example, after introducing three-way diffs, a diff group classified as an "add" relative to one source model may be classified as a "change" to the other. The visual representation will also require changes, either through introduction of additional colors or other visual markers to indicate the source model for each change, or by switching to a three-way, side-by-side presentation.

REFERENCES

[1] "Möbius website," accessed 2019. [Online]. Available: https://www.mobius.illinois.edu/

[2] G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. Webster, "The Möbius modeling tool," in *Proceedings 9th International Workshop on Petri Nets and Performance Models*, Sep. 2001, pp. 241–250.

[3] "Möbius wiki," accessed 2019. [Online]. Available: https://www.mobius.illinois.edu/wiki/index.php/M%C3%B6bius_Documentation

[4] D. S. Kolovos, R. F. Paige, and F. A. Polack, "Model comparison: A foundation for model composition and model transformation testing," in *Proceedings of the 2006 International Workshop on Global Integrated Model Management*, 2006, pp. 13–20.

[5] Y. Lin, J. Zhang, and J. Gray, "Model comparison: A key challenge for transformation testing and version control in model driven software development," in *Control in Model Driven Software Development. OOPSLA/GPCE: Best Practices for Model-Driven Software Development*. Springer, 2004, pp. 219–236.

[6] C. T. Buchanan, "Simulation debugging and visualization in the Möbius framework," M.S. thesis, University of Illinois at Urbana-Champaign, 2014. [Online]. Available: https://www.perform.illinois.edu/Papers/USAN_papers/14BUC01.pdf

[7] K. Keefe and W. H. Sanders, "Möbius shell: A command-line interface for Möbius," in *Quantitative Evaluation of Systems*, K. Joshi, M. Siegle, M. Stoelinga, and P. R. D'Argenio, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 282–285.

[8] E. A. LeMay, "Adversary-driven state-based system security evaluation," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2011. [Online]. Available: https://www.perform.illinois.edu/Papers/USAN_papers/11LEM02.pdf

[9] M. D. Ford, K. Keefe, E. LeMay, W. H. Sanders, and C. Muehrcke, "Implementing the ADVISE security modeling formalism in Möbius," in *Proc. 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013, pp. 1–8.

[10] "Views (Perforce documentation)," accessed 2019. [Online]. Available: https://www.perforce.com/manuals/v18.1/cmdref/Content/CmdRef/views.html

[11] "Git website," accessed 2019. [Online]. Available: https://git-scm.com/

[12] S. Derisavi, "The Möbius state-level abstract functional interface," M.S. thesis, University of Illinois at Urbana-Champaign, 2003. [Online]. Available: https://www.perform.illinois.edu/Papers/USAN_papers/03DER02.pdf

[13] "Eclipse Modeling Framework (EMF)," accessed 2019. [Online]. Available: https://www.eclipse.org/modeling/emf/

[14] W. H. Sanders and J. F. Meyer, "Stochastic activity networks: Formal definitions and concepts," *Lectures on Formal Methods and Performance Analysis: First EEF/Euro Summer School on Trends in Computer Science, Berg en Dal, The Netherlands, July 3–7, 2000 Revised Lectures, Lecture Notes in Computer Science*, pp. 315–343, 2001.

[15] "GEF (graphical editing framework)," accessed 2019. [Online]. Available: https://www.eclipse.org/gef/

[16] G. Clark and W. H. Sanders, "Implementing a stochastic process algebra within the Möbius modeling framework," in *Process Algebra and Probabilistic Methods: Performance Modelling and Verification: Proceedings of the Joint International Workshop, PAPM-PROBMIV 2001*, L. de Alfaro and S. Gilmore, Eds. Berlin: Springer, 2001, pp. 200–215.

[17] D. Koutra, A. Parikh, A. Ramdas, and J. Xiang, "Algorithms for graph similarity and subgraph matching," in *Proc. Ecological Inference Conference*, 2011. [Online]. Available: http://www.stat.cmu.edu/ aramdas/reports/DBreport.pdf

[18] A. Schipper, H. Fuhrmann, and R. von Hanxleden, "Visual comparison of graphical models," in *Proc. 14th IEEE International Conference on Engineering of Complex Computer Systems*, June 2009, pp. 335–340.

[19] K. Andrews, M. Wohlfahrt, and G. Wurzinger, "Visual graph comparison," in *Proc. 13th International Conference Information Visualisation*, July 2009, pp. 62–67.

[20] "EMF compare - developer guide," 2015. [Online]. Available: https://www.eclipse.org/emf/compare/documentation/latest/developer/developer-guide.html

[21] C. Brun and A. Pierantonio, "Model differences in the Eclipse Modeling Framework," *CEPIS UPGRADE*, vol. IX, No. 2, April 2008. [Online]. Available: http://www.cepis.org/upgrade/files/2008-II-pierantonio.pdf

APPENDIX A

EXPANDING GRAPHICAL COMPARISON SUPPORT FOR OTHER MÖBIUS FORMALISMS

The information provided in Chapter 4 focused on the general structure of data flow and processing needed for the new VCS feature as well as a few details of its implementation on the ADVISE formalism. For the purposes of integrating the VCS feature with future Möbius development efforts as well as with existing and future formalisms, more specific details and code will be given here. This appendix will be split into two sections: the first briefly describes changes related to high-level VCS operations and integration with Git, and the second, more extensive, section will provide information on how to extend support to other formalisms.

A.1 Git and VCS Integration

The current Git integration simply maps each option to an equivalent command executable on the command line. We have added new context menu options in the usual manner; each one maps to a newly created handler, which, in turn, runs the appropriate command. We created an additional version of the `ProjectImpl::runCmd()` function to facilitate redirection of command line output to a file for persistence. That mechanism is used primarily for creating temporary copies of past model versions used during model comparisons, but it may also be useful when new VCS commands are being added.

If support for another VCS is to be added, the problem of fundamental differences between the different VCS design philosophies presented in Section 2.1 will need to be addressed. As suggested in that section, the two most feasible solutions are either to build unique implementations of each VCS integration into Möbius or to map their commonalities onto a single set of unchanging options presented in the context menu to the user. In both cases, the existing Git handlers may be easily repurposed to execute different commands based on the VCS active in the selected project. It would be best if the VCS used by the project were also added as a configuration option in the global Möbius environment or were offered via a user prompt whenever a new project is created. A mechanism for detecting the active VCS in a Möbius project may also be needed; the VCS could be added to the project configuration

files, or the VCS could be obtained through scraping of the project directory for relevant metadata files.

A.2 Adding New Formalisms

The changes we made to implement support for graph comparison and merge operations on ADVISE are extensive. For clarity, they are presented here in three sections that detail newly added classes and their usage (Section A.2.1, changes made at the project component level A.2.2, and changes made to support visualization A.2.3.

A.2.1 New Classes

In the interest of presenting clean interfaces between internal components, we introduced several new classes into the core Möbius codebase. While it should not be necessary to change them in order to add support for a new formalism, it will be necessary to know how to use them.

1. `MobiusReferenceMerger` and `MobiusAttributeMerger`: These are extensions of the default merger object needed by the comparison algorithm implemented in *EMF Compare*. On its own, each of these behaves exactly like the default merger. However, they are intended to be used as base classes for formalism-specific mergers, and they each include a utility function for fetching the affected model element from the diff in question.

2. `ComponentDiffHandler`: This event handler is wired to the "Compare..." context menu option. When executed, it will first fetch the previous version of the selected `ProjectComponent` and call its `supportsGraphicalDiff()` function. If the result is "false," the Meld textual comparison tool will be launched to compare the newly fetched and current versions. Otherwise, the handler will run the model comparison pipeline detailed in Section 4.2 and open the modified model editor to display the results.

3. `DiffInfo`: This is effectively a tuple, packaging up each diff group (represented by its root) with the associated *kind* data and a reference to the affected model element. That reference is used to determine which graphical element should be highlighted when the visualization is being rendered.

4. `DiffList`: This class acts as a wrapper around a Java List, and provides a convenient interface through which one can build a single list of `DiffInfo` instances from multiple raw diffs.

5. `DiffMode`: This is a simple enumeration that lists the possible states of a model element. It is used to mark elements for visual highlight (see Section 4.3 during the model construction phase of the comparison pipeline. Each model element should contain an instance of the `DiffMode` attribute.

6. `ModelEObject`: Derived from the `EObject` base class shared by most of the EMF library, this interface is intended to be a base class for any class that implements a graph structure within a formalism. It defines several functions needed for model validation and visual refreshing. The ADVISE implementations for these functions may be found in the `AttackExecutionGraph` class.

A.2.2 Project Component Changes

This section details the changes we made to support model comparison for ADVISE. For clarity, function interfaces are intentionally omitted here. Additional comments and with specific details are available in the source code.

1. `AEGNodeReferenceDeletePreview`: This is a merger object (which inherits from the above `MobiusReferenceMerger` class) intended to replace the default `ReferenceChangeMerger` **only** for diffs that correspond to deleted references (see Section 4.2.1). Rather than carry out the default deletion behavior, this replacement should identify the affected model element and set its `DiffMode` accordingly. Since determination of the affected model element is dependent on the formalism, a version of this class will be needed for each supported formalism.

2. `AEGNode` and `Arc`: These classes have been updated to include a diffMode attribute as explained above.

3. `ProjectComponent` and `AdviseAtomicModel`: The base `ProjectComponent` class and the derived ADVISE variant have received several new functions. Some are self-explanatory and have been excluded from this list for brevity.

   3.1. `reopenWithDiffs()`: Reinitializes the `ProjectComponent` in "diff mode" and saves the input parameters.

3.2. `markElems()`: Determines the affected model element for each diff defined during initialization and sets the associated `DiffMode` appropriately. A copy of this value is also saved as part of the diff.

3.3. `getReferenceDeletePreview()`: Fetches the merger associated with the containing formalism as detailed above.

4. `AttackExecutionGraph`: This class now inherits from the `ModelEObject` class defined earlier and implements the required functions. Note that some part of the behavior defined in the following functions may be common and can be safely moved to the base class.

   4.1. `verifyModel()`: Performs model validation specific to each formalism. For ADVISE, it verifies that all `Arc` objects are connected on both ends. It also searches for and removes diffs that affect node attributes or arcs if the associated node has been removed from the model.

   4.2. `getRelatedDiffs()`: A Helper function that searches through a list of input diffs. If any are found that would remove node element, this function returns a list of diffs that affect attributes of that node.

   4.3. `remarkElems()`: Forces a reevaluation of the `ProjectComponent::markElems()` function. `remarkElems()` is used to refresh the visualization after a diff is resolved.

5. `SaveHandler` and `SaveNoCompileHandler`: The behavior of these handlers has been updated to redirect the output of a save command to the appropriate file when the command is executed on a preview model.

A.2.3 Visualization Changes

Changes we made specifically to support visualization of model comparison are detailed here. Since ADVISE is, at the time of this writing, the only formalism in Möbius that uses a graphical editor built on GEF, there is currently no inheritance structure available to support code sharing with other formalisms. Much of the work here can likely be moved into shared base classes once they are established.

1. `AEGNodePart` and `ArcPart`: The behavior of the color selection function for these visual parts is now dependent on the `DiffMode` of the associated model element.

2. `AdviseAtomicModelPart`: A new function has been added to this class that indicates whether or not the graphical editor is open in "diff mode." Furthermore, this class

is responsible for passing the relevant data to the `AEGComposite` from the back-end model.

3. `AEGComposite`: This class implements the `refreshVisuals()` function to reload the visualization after a merge change. It also instantiates an `AEGDiffComposite` when the editor is opened in "diff mode," as detailed below.

4. `AEGDiffComposite`: This composite replaces the standard node list in the graphical editor and displays the list of currently unresolved diffs. It is also an interface for handling merge operations on individual diffs.

5. `CanvasEditorComposite`: This class is responsible for the logic that supports merge operations via the `applyDiffs` and `undoDiffs` functions.