An Eclipse-Based Tool Framework for Software Model Management

Rick Salay Marsha Chechik Steve Easterbrook Zinovy Diskin Pete McCormick
Shiva Nejati Mehrdad Sabetzadeh Petcharat Viriyakattiyaporn
Department of Computer Science, University of Toronto
Toronto, ON M5S 3G4, Canada.
Email: {rsalay, chechik, sme, zdiskin, pete, shiva, mehrdad, apple}@cs.toronto.edu

ABSTRACT
Software development involves the use of many models and Eclipse provides an ideal infrastructure for building tools to support the use of models. While there is a large selection of tools available for working with individual models, there is less support for working with collections of models, as for example, when a collection of models from different sources must be merged. We have identified the problem of working with collections of related models in software development as the Software Model Management (SMM) problem - a close cousin of the Model Management problem in the area of metadata management. In the course of building SMM tools to address particular scenarios, we have observed that they share common foundations both at the theoretical and implementation levels. In this paper, we describe the vision and initial development of a framework that implements these common foundations in order to facilitate and accelerate the development of Eclipse-based SMM tools.

Categories and Subject Descriptors
I.6.5 [**Simulation and Modeling**]: Model Development

General Terms
Management, Design, Human Factors, Standardization, Languages, Theory.

Keywords
Model Management, Metamodeling, Multi-view Modeling, Model integration, Modeling Tools.

1. INTRODUCTION
1.1 Motivation
Software development has traditionally involved the use of many models and this is particularly the case with model-driven approaches. Thus, models are a fundamental type of artifact created and manipulated within Eclipse. To support this, core Eclipse provides an infrastructure for integrating multiple editors and tools within a single IDE as well as the GMF (Graphical Modeling Framework) as a model-driven approach to define editors for particular modeling languages. However, despite the strong support for individual models, Eclipse does not have an infrastructure for dealing with *collections of related models*.

Working with collections of models introduces special complexities because models are related and the integrity of these relations must be preserved. For example, in a distributed software development scenario, different teams may be responsible for different parts of a UML model but in order to specify the whole system, the relations between these parts must be expressed and then the parts must be merged into a single UML model in a way that correctly reflects these relations.

The area of metadata management has similar challenges due to the need to relate many schemas (i.e., models) in scenarios such as database integration, message mapping, data migration, etc. There, the field of Model Management [3] has emerged as a way to address these complexities by proposing that model relations be expressed as first class objects called *model mappings* and that generic operators be defined that could be used to manipulate models and mappings in a sound way to achieve various modeling goals. A key strength of this approach is a solid mathematical foundation [5].

Our research vision is to apply a similar approach to software modeling to support software development with many models. In particular, we are interested in investigating formally grounded approaches for expressing the relations between models (model mappings), defining operators such as *match* and *merge* to provide useful algebraic manipulations of models and mappings, defining methods for reasoning across multiple models and mappings, exploring ways to facilitate the comprehension of collections of related models, etc. We term this set of concerns Software Model Management (SMM) and distinguish it from but consider it complementary to MDE in that the former addresses issues in model-based development even within traditional software development paradigms, while the latter is primarily concerned with automating the process of model refinement toward the generation of code.

There have been various efforts to develop SMM tools, both within our group and elsewhere [13, 15, 8, 12], in order to address specific model management tasks and scenarios. Based on these experiences, we have observed that while building an SMM tool is difficult, there are common principles and infrastructure that underlie any such tool. These observations are the basis for initiating the Model Management Tool Framework (MMTF) project with the objective of extracting these common elements and providing a framework implementing them in order to simplify and accelerate the development of an Eclipse-based SMM tool. Specifically, the MMTF is intended to minimally address the following requirements:

- Support arbitrary model and model mapping types and operators over them.
- Support easy integration of existing independently developed model-related components including editors and operators.
- Provide the capability to import/create/modify/view particular collections of models and mappings.

- Provide the capability to interactively apply relevant operators to sets of models and mappings to derive new (resultant) models and mappings.
- Provide the capability to define new operators

Over time, we expect that this list will grow, and the MMTF will correspondingly be extended.

1.2 Related Work

The Eclipse-based Atlas Model Management Architecture (AMMA) [1] is a platform focused primarily on MDE and model transformation but has some components that widen this scope. The Atlas Model Weaver (AMW) provides a way to define model mappings while Atlas Megamodels (AM3) are metadata registries that relate resources such as models, metamodels and tools. While the MMTF provides some similar components, it focuses on the interactive and exploratory algebraic manipulation of models as part of model-based development rather than large scale model transformation infrastructures for MDE. Epsilon [7] provides a collection of model management task-specific languages and hence has a different and complementary objective to MMTF. Domain specific modeling language frameworks such as the Generic Modeling Environment (GME) [9] and MetaEdit+ [10] aim to provide a metamodel-configurable environment for producing model editors and transformation tools. As such, they can be seen as alternatives to GMF and transformation languages such as the Atlas Transformation Language (ATL). In contrast, the MMTF does not produce editors or transformations for particular modeling languages but instead works at a higher level of abstraction by integrating existing tools to facilitate the management of multiple models.

The approach taken by the MMTF is inspired in part by the work of Bernstein [2] on Model Management in the field of metadata management and although there are similarities between issues surrounding data schemas and models of software, there are significant differences as well. For example, the reason for expressing relations between two data schemas is typically to define a translation between their instances. On the other hand, with software models, the focus is on expressing relations to support activities like model merge or consistency checking. Nevertheless, the similarities between these areas is a potential source of foundational mathematics that we intend to exploit [4].

1.3 Organization of Paper

The paper is structured as follows. In Section 2 we discuss the functionality offered by MMTF to support the development of SMM tools. In Section 3, the architectural aspects of the implementation are described. Finally, in Section 4 we report on the current status of the project and our directions for the future.

2. FUNCTIONALITY

In [4], we identified and characterized a number of standard types of software model management operators including:

- *Merge*: combines two or more models with respect to known or hypothesized relationships between them.
- *Match*: finds commonalities between models, often as a preparation for merging them.
- *Diff*: identifies the (edit) distance between two models.
- *Split*: as the inverse to merge, partitions a model into views that have well-defined relationships between them.
- *Slice*: generates a partial view of a model, based on a stated criterion.
- *Check_property*: establishes whether a given property holds for a model, typically via model-checking.
- *Is_consistent*: establishes whether a set of models are semantically consistent according to the intended relationships between them.

These operators can be used individually or in combination to achieve various modeling objectives. For example, consider the following use case for SMM: In a distributed development scenario, different teams are developing StateChart models that address the portion of the system for which they are responsible. A lead architect is then responsible for integrating these into a single StateChart model of the system. She intends to use an SMM tool to help her do so.

An SMM tool for this use case would provide the following functionality:

- Allow StateChart models from the different teams to be imported into a common "workspace".
- Provide a way to define mappings that express the relations that the architect believes hold among the individual StateChart models. If she is uncertain about a particular relation, she may want to express alternative candidate mappings. Additionally, she may be supported by *Match* operators that propose different candidate mappings.
- Provide operators such as *Merge* that allow the architect to consider different combinations of the StateCharts models. Some examples of merge include those that include all behaviours that have been defined by *both* of the models, or those that have been defined by *either* of the models [11].

In our earlier work, we have built a special-purpose SMM tool called TReMer that supports this use case [15]. In order to help put the functionality offered by the MMTF in perspective, note that only about 30% of the TReMer code (e.g. the implementations of the *Match* and *Merge* operators) provides unique functionality. The rest can be factored out and is now provided by the MMTF. In the remainder of the paper, we will refer to this use case to illustrate how a re-implementation would proceed given the functionality and architecture of the MMTF.

2.1 Basic Functionality

The MMTF provides an environment into which different model and mapping types, editors and operators can be plugged in and integrated. In addition, we introduce a new type of model, called a Model Interconnection Diagram (MID). MIDs allow collections of models and mappings to be rendered graphically with nodes representing models and mappings between models. While a MID provides value as a useful level of abstraction in which to express a modeling scenario, the key role it plays is as an interface through which plugged-in functionality can be accessed. When a MID is opened using the MMTF MID editor, it provides the user with the ability to create model/mapping nodes for new or existing models/mappings, open model/mapping nodes by invoking the appropriate editor and manipulate models/mappings by invoking operators on collections of nodes. Figure 1 shows a screen shot of

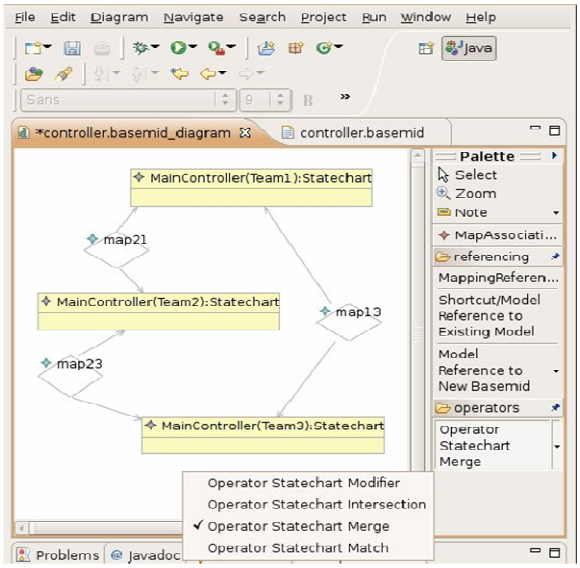

Figure 1. Screen shot of the MID editor.

the MID editor with an example MID from our StateChart scenario. Here, the user is about to apply a *Merge* operator to a selection of StateCharts and their mappings. Note that, for readability, the package explorer, outline and property panes are not shown. Also, in future versions of the editor we plan on rendering binary mappings as edges to simplify the presentation. We now consider various aspects of the MMTF functionality in greater detail.

2.2 Mappings

A *model type* is defined by a metamodel. Thus, instances of this metamodel are well-formed *models* of this type. Since the MMTF is a framework for Eclipse, the assumed default metamodeling language is Ecore + OCL for constraints; however, we intend to allow other languages as well (see Section 4).

A *mapping* is a special kind of model that is used to express the relationship between two or more models. Like any model, mappings are typed and are defined by a metamodel. Unlike simple models, however, they have external references to the models they relate and the elements within those models. A *mapping type* has a signature defining the model types it relates. For example, the mapping type `SCMapping` has signature:

```
SCMapping(StateChart, StateChart)
```

Thus, an instance of `SCMapping` is a mapping between two `StateChart` models.

The use of metamodels for mappings allows one to be flexible about what goes into a mapping. In addition, the semantics of a mapping can be partially captured, as with model types, by the well-formedness constraints. In particular, well formed mappings should be semantically sound relative to the intended semantics for the mapping. For example, a `SCMapping` includes state mapping and transition mapping elements. Furthermore, in a well formed `SCMapping`, if a transition t_1 in `StateChart` sc_1 is mapped to a transition t_2 in sc_2, then it should also map the corresponding endpoint states of t_1 to the endpoint states of t_2.

Since mappings have metamodels, they can be used wherever models are used. Thus, they can be arguments for operators, can have specialized editors and can be related by mappings as well.

Finally, certain generic mapping types can be defined based on the metamodels of the model types that they map. The notion of homomorphism between models of the same type is an example of this and it is of interest because it can be used to express a common class of mappings that have good algebraic properties. The `SCMapping` described above is, in fact, the homomorphism mapping type for `StateCharts`. The MMTF automatically defines the homomorphism mapping type for any model type and it provides a generic mapping editor that can be used with any homomorphism mapping type**.**

2.3 Operators

Like mapping types, operators are typed by a signature. For example, the *Match* and *Merge* operators described above have the following signatures:

```
SCMapping MatchSC(StateChart, StateChart)
StateChart MergeSC(StateChart, StateChart,
                   SCMapping)
```

An operator can be introduced either by using an operator plug-in or by an operator definition. In the latter case, the operator is defined as a composition of existing operators. For example, a combination operator `MatchAndMergeSC` could be defined as:

```
StateChart MatchAndMergeSC(
                   StateChart sc1,
                   StateChart sc2) {
    result = MergeSC(sc1, sc2,
                   MatchSC(sc1, sc2))
}
```

We are currently evaluating whether to define a new language for operator definitions or to adopt an existing one.

2.4 Diagrams

The MMTF follows the common distinction between diagrams (concrete syntax) and models (abstract syntax). A model is changed by modifying its diagram, and this is done using the editor for that type of diagram. Note that a model may have multiple diagrams associated with it, and these may be of different types. Thus, opening a model node requires that one of its diagrams be selected and the MMTF remembers this choice for subsequent openings. Furthermore, when an operator is invoked on a set of models/mappings, the framework provides the operator with access to the abstract syntactic and concrete syntactic information (i.e. diagrams) of these since they both may be relevant to the operator's behavior.

For example, a *Merge* operator that takes a set of related StateChart models and produces a single combined StateChart as the result should, at minimum, generate the abstract syntax of the result but ideally also one or more diagrams produced through "visually sensible" merges of diagrams of the component StateCharts.

2.5 Model Interconnection Diagrams (MID)

A MID captures and displays a collection of models and mappings between them. The metamodel in Figure 2 shows that a MID

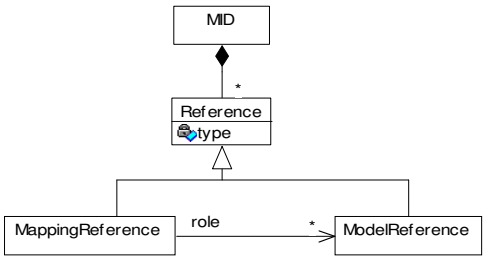

Figure 2. The metamodel for a MID.

consists of a set of references to the mappings and models it represents. When a MID is created or opened using the MID editor, it provides the user with the following options:

- Create a model node representing a new or existing model based on the plugged-in model type.
- Create a mapping node (or edge for binary mappings) representing a new or existing mapping based on the plugged-in or generic mapping types.
- Open a model node or a mapping edge/node and invoke the appropriate editor on the corresponding artifact using plugged-in or generic editors (such as with the homomorphism mapping type).
- Invoke an operator on a collection of models/mappings in the MID and view the result as an extension of the MID. Operators may be plugged-in or be user defined in terms of other operators.
- View the metadata associated with a model/mapping. This includes information about the history of operator applications that produced it or in which it was involved.

Since mappings, operators and editors are strongly typed, the MID editor constrains their applicability to models of the appropriate types. As part of future work, we are investigating a richer typing mechanism that would allow subtyping and type conversion to allow broader applicability (see Section 4).

3. ARCHITECTURE

Figure 3 depicts the architecture of the MMTF. The primary mechanism for building an SMM tool using the MMTF is to provide plug-ins defining model/mapping types, their editors and operators over them. Each of these types of plugins interact with the MMTF services via extension points and interfaces published by the MMTF. Since we are intending to leverage and extend

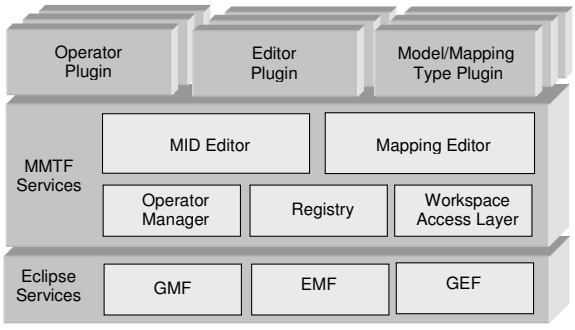

Figure 3: MMTF Architecture.

Eclipse and GMF, we further make the following assumptions. We assume that Model/Mapping types have an associated Ecore metamodel and a corresponding EMF plug-in generated from it. In addition, we assume that Editor plug-ins are GMF-based editors and hence base their diagrams on the GMF notation metamodel.

In the MMTF services, the MID editor and the Mapping Editor are GMF-based editors that implement the functionality described in Section 2. The MID editor adapts to the set of plug-ins by providing palette and context menu entries that allow the creation of nodes for plugged-in model/mapping types, the opening of nodes using plugged-in editors, and the application of plugged-in operators.

A basic requirement of the MMTF is that it should act as an integration point for independently developed components and content. This requirement is supported by the following architectural features:

- The metamodels in Model Type plug-ins are independent and are related only via the metamodels of Mapping Type plug-ins.
- Editor and Operator plug-ins are dependent only on the plug-ins of the model/mapping types they address.
- Models and mappings are persisted as independent XMI files even though they may appear to be "gathered together" within a particular MID. All references to models or mappings are via Uniform Resource Identifier (URI).

In the MMTF services, the Registry and the Workspace Access Layer use the façade pattern to provide integrated access to the components and content, respectively. Figure 4 depicts the object model. It provides metadata about components and content and decouples their use from their implementation details. Thus, for example, an operator may be implemented as a plug-in or using our operator definition language and this is transparent to consumers of operators. Similarly, the use of alternate repositories for models and mappings can be implemented without affecting existing mappings or MIDs that reference them. Currently, we do not support dynamic plug-ins, so Registry contains only static information about plug-ins that the tool is running; however, the Workspace Access Layer remains synchronized with the open workspace and listens for changes in the file system.

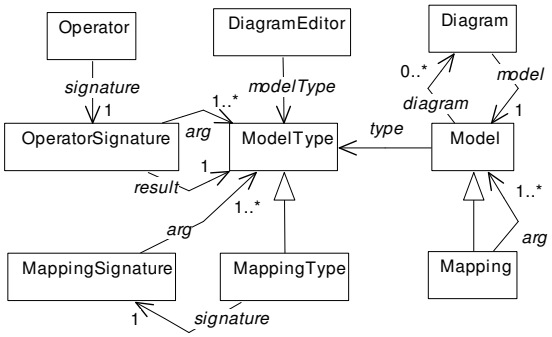

Figure 4: Object model of Registry and Workspace Access Layer.

The Operator Manager is responsible for determining operator applicability to a given set of models/mappings as described in Section 2 and for managing the invocation of the operator. Operator invocation either involves delegation to a plugged-in operator or the execution of a user-defined operator. In either case, an operator is treated like a command and uses the GMF command framework to provide support for capabilities such as undo/redo and progress monitoring.

4. CURRENT STATUS AND FUTURE DIRECTIONS

We have completed the first phase of development on the MMTF. This includes basic support for model, editor and operator plug-ins as well as first versions of the MID editor, Registry, Workspace Access Layer and Operator Manager. The second phase is expected to be complete by the time this paper is presented and includes support for mapping types, generic Mapping Editor and operator definitions.

For subsequent phases we are exploring the following possibilities:

- **Alternative Metamodeling Languages**: We assume that model/mapping types are defined using Ecore+OCL metamodels; however, in the research context we also want to consider other metamodeling languages. We are exploring ways to support this.
- **Type Hierarchies and Type Conversion**: Model/mapping types can be organized into a type hierarchy. Semantically, this means that an instance of a more specialized type is also an instance of a more general type and thus could be used as an argument to an operator or an argument of a mapping type defined for the more general type. Another possibility is to use an editor for the more general type as a viewer (i.e. read-only) for the more specialized type. We are studying how to implement these cases by expressing the subtype relationship between metamodels in a way that correctly relates the well-formedness conditions and then doing automatic type inferencing in the MID editor. In addition, since subtyping could be seen as a special case of type conversion, we are investigating this more general case as well.
- **MID-based Operators**: Currently an operator signature restricts operators to accept a fixed tuple of models/mappings; however, we may want to allow operators that could accept arbitrary collections of models and mappings. For example, a more general `MergeSC` operator should be able to accept a set of `StateCharts` and `SCMappings` and produce a single `StateChart` as the result. To do this, we need to define typed collections of models/mappings – i.e. typed MIDs. A MID type is given by a set of model and mapping types and an instance MID can only contain models/mappings of these types. Operator signatures for typed collections can be defined in terms of MID types.
- **Category Theory-Based Engines**: Category Theory provides a formal basis for many concepts within model management [5]. We are exploring ways to incorporate this theory into the MMTF in order to provide more generic services to the tool builder. For example, Category Theory defines structure composition operations known as *limit* and *colimit* in a very general way and these could be used to implement a *Merge* operation for models of any type. We have already exploited this in other work [14, 6] and seek to build it into the MMTF as well.

5. ACKNOWLEDGMENTS

This work is supported, in part, by an NSERC grant and an IBM Eclipse Innovation Grant.

6. REFERENCES

[1] Atlas Model Management Architecture home page: http://www.sciences.univ-nantes.fr/lina/atl/AMMAROOT/

[2] Bernstein, P.: Applying Model Management to Classical Meta Data Problems. In *Proc. Conf. on Innovative Database Research*, pp. 209-220, 2003.

[3] Bernstein, P.A., Melnik, S. Model Management 2.0—Manipulating Richer Mappings, *Proc. SIGMOD 2007*, pp. 1-12.

[4] Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N., Sabetzadeh., M. A Manifesto for Model Merging, In *Proc. 1st Int. Workshop on Global Integrated Model Management* (associated with ICSE'06), May 2006.

[5] Diskin, Z. Mathematics of Generic Specifications for Model Management I and II. In *Encyclopedia of Database Technologies and Applications*. Idea Publishing Group, pp. 351-366, 2005.

[6] Diskin, Z., Dingel, J. and Liang, H. Scenario Integration via Higher-Order Graphs. Technical Report No. 2006-517 School of Computing, Queen's University, 2006.

[7] Epsilon tool page: http://www.eclipse.org/gmt/epsilon/

[8] Fleurey, F., Baudry, B., France, R., Ghosh, S. A Generic Approach For Automatic Model Composition. In *Proc. AOM at MoDELS '07.* 2007.

[9] Ledeczi A., Maroti M., Bakay A., Karsai G., Garrett J., Thomason IV C., Nordstrom G., Sprinkle J., Volgyesi P. The Generic Modeling Environment, In *Proc. Workshop on Intelligent Signal Processing*, May 17, 2001.

[10] MetaEdit+ homepage: www.metacase.com

[11] Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., Zave, P. Matching and Merging of Statecharts Specifications. In *Proc. ICSE'07*, pp.54-64, May 2007.

[12] Ohst, D., Welle, M., and Kelter, U. 2003. Differences between versions of UML diagrams. *SIGSOFT Softw. Eng. Notes* 28, 5 (Sep. 2003), 227-236.

[13] Sabetzadeh, M, Easterbrook, S.M. iVuBlender: A Tool for Merging Incomplete and Inconsistent Views. In *Proceedings of Int. Conf. on Requirements Engineering* (RE'05), pp. 453-454, Sept. 2005.

[14] Sabetzadeh, M, Easterbrook, S. View Merging in the Presence of Incompleteness and Inconsistency. *Requirements Engineering Journal*, 11(3), pp. 174-193, June 2006.

[15] Sabetzadeh, M., Nejati, S., Easterbrook, S., Chechik, M. A Relationship-Driven Framework for Model Merging. In *Proc. Int. Workshop on Modeling in Software Engineering (MiSE'07)* at ICSE'07, 2007.