MMINT: A Graphical Tool for Interactive Model Management

Alessio Di Sandro * Rick Salay * Michalis Famelis * Sahar Kokaly † and Marsha Chechik *
* University of Toronto, Canada
Email: {adisandro, rsalay, famelis, chechik}@cs.toronto.edu
† McMaster University, Canada
Email: kokalys@mcmaster.ca

***Abstract*—Model Management addresses the accidental complexity caused by the proliferation of models in software engineering. It provides a high-level view in which entire models and their relationships (i.e., mappings between models) can be manipulated using transformations, such as *match* and *merge*, to implement management tasks. Other model management frameworks focus on support for the programming required to prepare for model management activities at the expense of the user environment needed to facilitate these activities. In this paper, we demonstrate *MMINT*- a model management tool that provides a graphical and interactive environment for model management. *MMINT* has many features that help manage complexity while reducing effort in carrying out model management tasks. We illustrate these features with a realistic model management scenario. (See video at: http://youtu.be/7B7YuV-Jvrc)**

I. Motivation and Goals

Modern software development requires managing multiple, heterogeneous software artifacts, but the proliferation of models creates an accidental complexity that must be managed. Model Management [1] addresses this challenge. This is realized through the use of special models called *megamodels* [2] to represent model management scenarios at a high level of abstraction and general purpose transformations (also called *operators*) [3] to manipulate models and their relationships (i.e., mappings between models) in order to perform useful tasks. For example, a model *match* operator finds correspondences between two models and packages these as a mapping between the models. A *merge* operator can be used to combine the content of the two models using the correspondence information in the mapping.

Several model management frameworks have been developed to support Model Driven Engineering (MDE) by facilitating the development of operators and the implementation of specific model management tasks. For example, Epsilon [4] provides multiple domain-specific languages, each specialized for a different model management task such as merge, validation, transformation, etc. The Atlas suite of tools [5] is centred around the ATL transformation language and its use in different model management tasks. These frameworks focus on the *programming required to prepare for model management* rather than the *user environment required to carry out model management tasks*. However, such an environment is a key factor for practical model management.

An effective user environment requires a rich *interactive user interface* and *automated user assistance*. We elaborate on these in the context of model management:

- *Interactive User Interface* - Megamodels provide the means for raising the level of abstraction to manage complexity in model management, thus they are the natural user interface for model management. Furthermore, the user should be able to interact with the *type level* where model types (i.e., metamodels), relationship types and transformations are defined as well at the *instance level* where particular model management scenarios are expressed and carried out. *Interactivity* is an important characteristic of the user interface. Due to the complexity of model management tasks, they often can only be partially automated. For example, performing a match between models may need to be manually verified to ensure the correspondences are sensible, and performing a merge of models may yield conflicts that need to be manually resolved using human judgment.
- *Automated User Assistance* - Many model management activities are one-off tasks that suggest the need for low effort rapid implementation, possibly with manual steps, rather than a polished and fully automated process. Even activities that become fully automated often initially require experimentation and exploration to get them right. User assistance to help adapt transformations for reuse, support reasoning about models, etc. can reduce effort and help manage complexity.

In this paper, we describe our tool called *MMINT* (*"Model Management INTeractive"*) for graphical, interactive model management. We summarize the features of *MMINT* addressing the requirements discussed above.

- *Interactive User Interface*
  - *instance level megamodel* - *MMINT* provides a customizable environment in which modellers can graphically create model management scenarios using megamodels at the instance level, interactively apply transformations and immediately see their result and, when necessary, manually drill-down into the detailed content of particular models or relationships.
  - *type level megamodel* - a megamodel is used at the

type-level to visualize and allow rapid modifications of the type hierarchy of model types, relationship types and transformations at run-time to immediately impact instance level megamodels.

- *Automated User Assistance*
  - *megamodel operators* - the generic operators *map*, *filter* and *reduce*, available in many programming languages to simplify complex collection-based manipulation tasks, are built-in and adapted for use with megamodels [6].
  - *type coercion* - a type coercion mechanism is available that automatically performs the necessary conversion transformations required to reuse transformations from related types.
  - *retyping* - a type down-casting mechanism is available that automatically detects cases when a model satisfies the constraints of a more specialized type and thus can use its transformations.

The rest of this paper is organized as follows: In Sec. II we introduce and illustrate the features of *MMINT* by implementing a detailed model management scenario. In Sec. III, the architecture of *MMINT* is described. Finally, in Sec. IV, we conclude with a summary of the paper and a discussion of future research directions.

II. Using *MMINT* for Model Management

In this section, we use a model management scenario to illustrate the *MMINT* features addressing the requirements for a model management user environment discussed in Sec. I.

A. Illustrative scenario

Consider the following model management scenario. A company uses a megamodel to track its modeling artifacts (models and relationships between them). The company has determined that having public attributes in class diagrams is undesirable and now (1) would like to identify all class diagrams that contain this construct, (2) refactor them using a predefined transformation to remove the construct, (3) merge the modified class diagrams with the originals into a single class diagram, and finally, (4) produce a textual representation of the merged class diagram.

We now show how *MMINT* can be used to accomplish this scenario. Table I summarizes the *MMINT* features and the step of the scenario in which they are illustrated.

TABLE I
MMINT FEATURES AND WHERE THEY ARE ILLUSTRATED IN THE SCENARIO.

Requirement	*MMINT*	Step
Interactive User Interface	Instance megamodel	All
	Type megamodel	1
Automated User Assistance	Megamodel operators	
	map	2,3
	filter	1
	reduce	3
	Type coercion	4
	Retyping	1

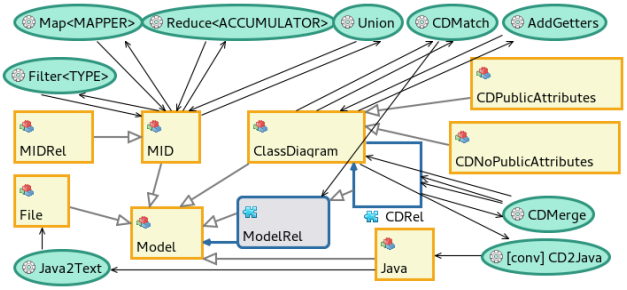

Fig. 1. Type megamodel in *MMINT* used for the examples in this paper.

B. Executing the scenario

In *MMINT* a megamodel is referred to as a *MID* (Model Interconnection Diagram). *MMINT* uses a distinguished *Type MID* in which model types, relationship types and transformations are registered. Fig. 1 shows a screenshot of the Type MID used to implement the scenario. Here, boxes represent model types and thick blue links between them are binary relationship types. The sub-typing between types is shown with the hollow-headed arrows. Transformations are ovals connected to their input and output types with named links (not shown to avoid clutter). For example, our scenario needs class diagrams and so `ClassDiagram` is a model type that is a sub-type of the general model type `Model`. The binary relationship type `CDRel` is a used to create mappings between `ClassDiagram` models. `CDMatch` is a transformation that takes two `ClassDiagrams` as input and produces a `CDRel` between them as output that contains links between classes that have the same name. Note that even the built-in megamodel operators `Filter`, `Map` and `Reduce` appear as transformations in the Type MID.

At the instance level, *MMINT* is used to create MIDs using a *MID editor* that allows an engineer to interactively create or import models, relationships and other MIDs, invoke transformations on them and inspect or change the results. We will incrementally and interactively build a MID called `Scenario` as we carry out the steps of the scenario. Fig. 2 (top-right) shows a screenshot of the final state of this MID after step (4). Initially, `Scenario` contains only the top left-most box referring to MID `Lib` (top-left of Fig. 2) that holds the class diagrams the company wants to process. We have limited the the number of CD's in MID `Lib` to four for this illustration; however, the scenario scales well (See [6]).

Step (1). In the first step we need to separate out the class diagrams in `Lib` that contain public attributes. We can use the megamodel operator `Filter`⟨`TYPE`⟩ that, when applied to a MID, removes all models/relationships that do not conform to type `TYPE` (See [6] for details); however, we need a model type that represents class diagrams that contain public attributes. The Type MID allows an engineer to create new types and use them immediately. Thus, we use the Type MID editor to define a new sub-type `CDPublicAttributes` of `ClassDiagram` containing the following OCL well-formedness constraint:

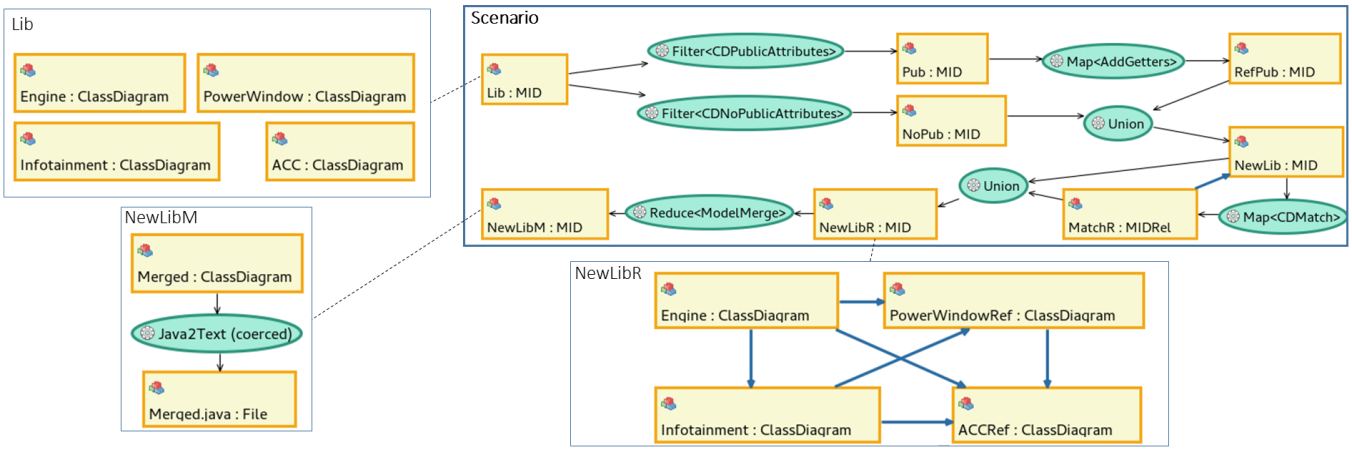

Fig. 2. Screenshot of the final state of MID `Scenario` and selected other MID's used or created in the scenario.

```
CDPublicAttributes:
  Attribute.allInstances()->exists(
    attribute | attribute.public)
```

Similarly, we create sub-type `CDNoPublicAttributes` with the negation of this constraint. Both are shown in Fig. 1 as sub-types of `ClassDiagram`. We then split `Lib` into new MIDs `Pub` (containing CD `PowerWindow` and `ACC`) and `NoPub` (containing CD `Engine` and `Infotainment`) by applying `Filter⟨CDPublicAttributes⟩` and `Filter⟨CDNoPublicAttributes⟩`, respectively, to `Lib`.

This step also illustrates the *retyping* feature of *MMINT*. Even though the class diagrams in `Lib` are typed as `ClassDiagram`, *MMINT* can check them against a sub-type (e.g., `CDPublicAttributes`) and automatically retype them if they conform. This feature can be invoked manually on any model by the engineer or internally by other operators – in this case, `Filter`. When invoked manually on a model (via right-clicking and a context menu), *MMINT* traverses the type hierarchy to check the model's conformance with all sub-types and returns the list of retyping options for selection by the user.

Step (2). In this step, we use a standard class diagram refactoring transformation, called `AddGetters`, that replaces each public attribute with a private attribute and a corresponding getter method. *MMINT* allows transformations in any language to be added to the Type MID. In this case, we have used Henshin [7] to implement `AddGetters`. We want to apply `AddGetters` to every class diagram in MID `Pub`. We can achieve this by using the megamodel operator `Map⟨MAPPER⟩` that applies a transformation given by parameter `MAPPER` to each model/relationship in a MID (See [6] for details). In this case, we apply `Map⟨AddGetters⟩` to `Pub` to produce MID `RefPub` containing the refactored class diagrams. We then use the built-in operator `Union` to produce MID `NewLib` that combines the MID `RefPub` with the class diagrams containing no public attributes in MID `NoPub`. Thus, `NewLib` is the same as the original MID `Lib` but with refactored class diagrams.

Step (3). Now we wish to merge the class diagrams in `NewLib` into a single class diagram. We have available to us a class diagram merge transformation `CDMerge` that takes two class diagrams and a `CDRel` relationship between them indicating how the class diagrams overlap [3]. That is, if the relationship has a link between two classes then `CDMerge` will combine these classes into a single class in the output merged model. Unfortunately, our MID `NewLib` contains only class diagrams and no relationships indicating their overlap. We can fix this by first using the transformation `CDMatch` that creates a `CDRel` between two class diagrams and puts links between classes with the same name. We could apply `CDMatch` manually to each pair of class diagrams in `NewLib` but to save time, we apply `Map⟨CDMatch⟩` to `NewLib` and get all the relationships in one step in the MIDRel `MatchR`. A *MIDRel* is a megamodel containing only relationships. Finally, we union these relationships with the class diagrams to produce MID `NewLibR` (content shown in Fig. 2).

At any point in our scenario we can drill-down into models or relationships to examine results or manually intervene in the process. For example, we may want to manually check or modify individually relationships that `Map⟨CDMatch⟩` produced to confirm that the correct classes are linked. To do this, we would double-click on a relationship (thick blue arrow in `NewLibR`) and view or edit it using the *MMINT* `relationship editor`.

Now we can merge the models in `NewLibR` pairwise using `CDMerge`. To do this quickly we use the megamodel operator `Reduce⟨ACCUMULATOR⟩` that accepts an arbitrary "merging" transformation and keeps applying it until it can be applied no more (See [6] for details). In this case we apply `Reduce⟨CDMerge⟩` to MID `NewLibR` to produce the MID `NewLibM` containing a single class diagram with the merged content of the class diagrams in `NewLibR`.

Step (4). In the final step of our scenario we wish to create a Java representation of the the merged class diagram in MID

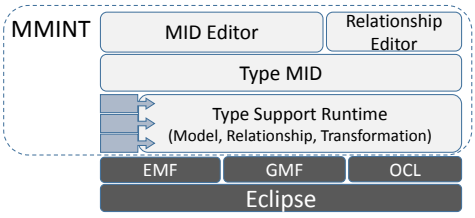

Fig. 3. Architecture of *MMINT*.

`NewLibM`. Assume we already have a transformation `CD2Java` that converts a class diagram to a Java model (i.e., based on a Java metamodel). In addition, we have a transformation `Java2Text` that produces the textual code of a Java model. One way to get the result we want is to apply these transformations in sequence on the merged class diagram in MID `NewLibM`. However, this step can be simplified by using the *MMINT* feature of *type coercion* that enables transformation reuse.

MMINT allows some transformations in the Type MID to be explicitly marked as *conversion* transformations and this is the case with `CD2Java` (See Fig. 1). Normally, when a model is selected in a MID and right-clicked, *MMINT* computes the list of applicable transformations based on the Type MID and presents a list to the engineer to choose from. The type coercion features means that *MMINT* not only lists the transformations that directly apply to the model based on its type, but also transformations that could indirectly apply if the model was first converted using conversion transformations and it does the conversions automatically. In our scenario, this means that when we right-click on the merged class diagram in MID `NewLibM`, the transformation `Java2Text` will be available and if selected, *MMINT* runs the necessary `CD2Java` conversion behind the scenes and then deletes the intermediate result.

III. *MMINT* Architecture

MMINT[1] is an evolution of the MMTF model management framework [8] adding the Type MID, support for retyping, type coercion and megamodel operators. *MMINT* is implemented in Java and uses the Eclipse Modeling Framework (EMF) [9] to express models and the Eclipse Graphical Modeling Framework (GMF) to create custom editors for editing models and relationships. The overall architecture of *MMINT* is illustrated in Fig. 3.

Model and Relationship types. Metamodels for new types can be plugged in or created directly through the Type MID. Implementations for supporting tools such as type-specific editors, validation checkers and solvers can also be plugged in and are managed by the type support runtime layer. A generic relationship editor is built into *MMINT*.

Transformations. New transformations can be implemented in Java or any transformation language and plugged into the type support runtime layer as transformation definitions. The definition includes metadata specifying the input and output types and other attributes such as whether this is a conversion transformation used by the coercive typing feature. Transformations can also be designated as higher order and take types or transformations as parameters. The megamodel operators `Filter`, `Map` and `Reduce` are implemented in this way. Transformation metadata is stored in the Type MID (See Fig. 1) for use at run-time.

MID Editor. The MID Editor provides the user interface through which MIDs are created and manipulated. Double-clicking on an element (model, relationship or MID) opens the corresponding artifact using the appropriate editor. Right-clicking on an element brings up a context menu that provides functionality appropriate to the corresponding artifact including retyping assistance, transformations that can be applied (including indirect ones via coercion) and validation.

[1]Available at: http://github.com/adisandro/MMINT

IV. Conclusion and Future Work

Existing model management frameworks focus on providing programming tools in preparation of model management, rather than providing a user environment in which to carry out model management tasks. *MMINT* allows users to perform automatically assisted model management tasks through the use of interactive type and instance megamodels. We have illustrated this in a detailed model management scenario.

In the future, we intend to extend *MMINT* with a formal model management workflow language, that allows users to construct verifiable model management scenarios as chains of declarative model management operations (e.g., model translation, model merge) with the use of predefined workflow constructors (e.g., sequential composition, branching, etc.). We are also investigating how to integrate *MMINT* with other model management approaches.

References

[1] P. A. Bernstein, "Applying Model Management to Classical Meta Data Problems," in *Proc. of CIDR'03*, vol. 2003, 2003, pp. 209–220.
[2] J. Bézivin, F. Jouault, and P. Valduriez, "On the Need for Megamodels," in *Proc. of OOPSLA/GPCE Workshops*, 2004.
[3] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh, "A Manifesto for Model Merging," in *Proc. of GAMMA at ICSE'06*, 2006, pp. 5–12.
[4] D. S. Kolovos, L. M. Rose, A. Garcia-Dominguez, and R. F. Paige, *The Epsilon Book*. Eclipse, 2015.
[5] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez, "Modeling in the large and modeling in the small," in *Model Driven Architecture*. Springer, 2005, pp. 33–46.
[6] R. Salay, S. Kokaly, A. Di Sandro, and M. Chechik, "Enriching Megamodel Management with Collection-Based Operators," in *Proc. of MODELS'15 (To appear)*.
[7] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, "Henshin: Advanced Concepts and Tools for In-place EMF Model Transformations," in *Proc. of MODELS'10*, 2010, pp. 121–135.
[8] R. Salay, M. Chechik, S. Easterbrook, Z. Diskin, P. McCormick, S. Nejati, M. Sabetzadeh, and P. Viriyakattiyaporn, "An Eclipse-Based Tool Framework for Software Model Management," in *Proc. of Eclipse Workshop @ OOPSLA'07*, 2007, pp. 55–59.
[9] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.