Using a Model Merging Language for Reconciling Model Versions

Klaus-D. Engel[1], Richard F. Paige[2], and Dimitrios S. Kolovos[2]

[1] Fraunhofer FOKUS
engel@fokus.fraunhofer.de
[2] Department of Computer Science, University of York, UK
{paige, dkolovos}@cs.york.ac.uk

Abstract. A difficult challenge in the industrialisation of Model-Driven Development is managing different *versions* of models. Different versions may arise at any time during the development process, due to different individuals or teams working on different parts of the overall model. To manage these versions it is necessary to be able to identify differences and *reconcile* these differences in a single, integrated model. We describe the use of *model merging* technology for managing different versions of a model in an industrial software development process. The use of automated model merging technology is contrasted with an alternative, semi-automated approach. The contributions of model merging to helping to solve this problem are outlined.

1 Introduction

Industrial applications of Model-Driven Development (MDD) require facilities for managing models. In particular, such facilities need to offer support for creating, updating (e.g., transforming or merging models), analysing (e.g., consistency checking) and deleting a variety of models, manipulated by different stakeholders. Model management platforms are beginning to become available – such as the AMMA [6] and Epsilon [7] platforms.

A key problem faced by developers of model management platforms is the ability to manage and manipulate different *versions* [11] of the same model. This is particularly relevant in an industrial application of MDD where different developers work on the same model at different times: differences and similarities between different model versions need to be *identified*, *reconciled*, and finally *integrated* into a unified single model. This is a difficult problem in itself, given the complexity of industrial MDD models, and the variety of changes that can be made to them, in a typical MDD process. The problem is exacerbated in the case where a solution to model version management needs to be introduced, *without* changing the existing development process, e.g., by replacing a predominantly manual solution with an automated or semi-automated one.

In this paper, we sketch an approach to managing model versions based on use of a model merging language – the Epsilon Merging Language (EML). In order to properly assess the value and utility of applying model merging in this context, we

A. Rensink and J. Warmer (Eds.): ECMDA-FA 2006, LNCS 4066, pp. 143 – 157, 2006.

aim to compare the use of EML with an alternative, semi-automated solution – developed in-house – that did not use model merging facilities.

The remainder of this paper is as follows. We first briefly define the problem of managing model versions, and touch on related work. We introduce the Epsilon Merging Language, and then provide context by explaining the development process into which model version management is to be introduced. Two solutions to managing model versions are presented: the first not based on model merging, and the second using model merging via EML. The two approaches are then compared, and lessons learnt identified.

2 Background and Related Work

We start with a brief overview of the characteristics of a version control/management system, and then discuss model version control. We then give a short overview of the Epsilon Merging Language (EML), focusing on its characteristics that can help with model versioning.

2.1 Version Control and Model Versioning

A version control system (also called a *revision control system*) [14] is a software system that manages multiple revisions of the same unit of information. Conceptually, any serialisable information can be managed by a version control system (VCS). In practice, a VCS is typically applied to source code and textual documentation. Other applications include managing versions of CAD files. Some modelling tools, e.g., Enterprise Architect [8], include built-in support for managing versions of diagrams, but do not offer all of the facilities of a fully-fledged VCS.

The fundamental characteristics of a VCS as applied in software engineering are as follows [14]:

- the ability to return to any earlier state in the design (e.g., rollback to a previous version of a source file because of the introduction of a bug).
- to allow multiple versions of a software system to be executed independently (e.g., to identify in which version a bug arose).
- to allow multiple developers to work on a system simultaneously.
- to allow documentation of changes and revisions (e.g., the changes that were made in moving from one version to the next).
- to allow identification of differences between versions.
- to allow developers to merge different versions.

There are both centralised repository and distributed variants of VCSs, which mainly differ in terms of the approaches taken to avoiding conflicts (e.g., locking mechanisms).

A key difficulty in using traditional version control systems for managing versions of models [11] is that the traditional approaches are based on linear, text-based files.

Models, however, are structured graphs presented visually. Traditional VCSs are not designed to operate with hierarchical data.

The key issues in resolving this abstraction mismatch are identified by Lin, Zhang and Gray [10], who clarify the need for efficient and precise definitions of *model comparison*, needed for supporting model version control. Alanen and Porres [12] formalise a definition of union and difference of models which can form the basis of an implementation of model version control, perhaps based on XML or XMI serialisation [3] of models. One challenge to overcome is the problem of visualising model differences. Ohst et al [9] make use of colours to highlight differences and redundancies, but it is unclear whether this is sufficient to cover all the potential differences (e.g., between elements and between hierarchies), and whether a colour-based approach scales.

2.2 Epsilon Merging Language

The Epsilon Merging Language (EML) is a metamodel-independent language for expressing model composition operations. It is built atop a generic model management language called the Epsilon Object Language (EOL) [5], which is inspired by OCL. EML is a general-purpose model composition language, and is rule based. It allows specification of three different kinds of rules and fulfills the general requirements for a model comparison solution identified in [9]. EML also supports model transformations, by building atop the EOL. EML supports model transformations by *transform rules*. EOL provides only model navigation and generic management of models.

An EML specification consists of a set of rules describing how model compositions should be carried out. Rules in EML are of three types:

- Match rules
- Merge rules
- Transform rules

Match rules can be further subdivided into comparison and conformance rules (examples to follow). EML also provides support for a *pre* block and a *post* block, which are actions that are executed prior to and after the compositions have taken place. These blocks are used to perform tasks that are not pattern based (e.g., initialisation and post-processing; an example will follow).

Each match rule has a unique name and two metaclass names as parameters. The rule itself is composed of a *compare* part and a *conform* part . The rule is executed for all pairs of instances of the specified metaclasses that appear in the source models.

The compare part of a match rule determines whether two instances match, using a minimum set of (syntactic) criteria. The conform part applies only to instances that satisfy the compare part of a rule; the conformance rule set refines this match. If the conformance part of the rule fails, then an exception is raised (work is ongoing on improving EML's exception handling capabilities).

An example is shown in Figure 1.

```
abstract rule ModelElements
     match l: Left!ModelElement
     with r: Right!ModelElement
     extends Elements {

     compare {
            return l.name = r.name
            and l.namespace.matches(r.namespace);
     }
}

rule Classes
     match l: Left!Class
     with r: Right!Class
     extends ModelElements {

     conform { return l.isAbstract = r.isAbstract; }
}
```

Fig. 1. Matching rules in EML

The rule ModelElements is abstract; it is not instantiated and is not used to carry out any matches. It provides basic behaviour used by rules that *extend* it. The basic behaviour of this abstract rule is to match model elements that have identical names (l.name=r.name) and matching namespaces. A similar match rule is used for classes. However, the Classes rule is concrete and will be executed by the EML virtual machine. Classes match when they obey the rules declared in their parent, the rules they extend (in our case the ModelElements rule), and when the additional *conform* part of the rule holds, i.e., when classes are either both abstract or both not abstract.

2.2.1 EML Model Element Categorisation

After the execution of all match rules in an EML specification, four types of model elements can be identified with respect to a particular pair of models (which we designate as *left* and *right* models, respectively)

1. Elements that *match and conform* to elements of the opposite model (i.e., elements of the left model that match and conform to elements of the right model, and vice versa).
2. Elements that *match but do not conform* to elements of the opposite model. Elements in this category trigger cancellation of the composition process.
3. Elements that *do not match* with any elements in the opposite model.
4. Elements on which no matching rule has applied; elements in this category may suggest that the specification is incomplete and thus trigger warnings.

After the matching rules have been applied, the following results are obtained.

- Elements that match and conform will be merged with their identified opposites. The specification of merging is captured in a *merge rule*.
- Elements in categories 3 and 4 (that do not match) will be *transformed* into model elements compatible with the target metamodel. The specification of transformation is defined in a transformation rule. We do not go into further details regarding transformation rules in this paper.

- Elements in category 2 either generate or an exception, or are handled by a *fix* block (similar to *try-catch* in Java), which we discuss further in Section 3.

2.2.2 EML Merge Rules

Merge rules in EML are used to specify the behaviour necessary to compose two instances of model elements that match and conform. Each merge rule consists of a unique name, two metaclass names as input parameters, and the metaclass of the model element that the rule creates in the target model.

For all pairs of matching instances of the two metaclasses, the rule is executed and an empty model element is created in the target model. The content of the newly created model element is defined by the body of the merge rule. Two examples of merge rules are shown in Figure 2.

```
rule ModelElements {
   merge l: Left!ModelElement
   with r: Right!ModelElement
   into m: Merged!ModelElement

   m.name := l.name;
   m.namespace:=l.namespace.equivalent()
}
```

```
rule Classes {
   merge l: Left!Class
   with r: Right!Class
   into m: Merged!Class
   extends ModelElements {

   m.feature := l.feature.
     includeAll(r.feature).
     equivalent();
 } }
```

Fig. 2. EML Merge Rule

Figure 2 presents two merge rules, one for merging ModelElements and a second for merging UML classes ("Classes"). The first rule applies to all Model Elements and produces a new, merged ModelElement whose name is that of the left original model, and whose namespace is that of the left original model. In the second rule, the two metaclasses, *left* and *right*, are declared; the merge rule is also declared to produce an instance of Class metaclass. The result of applying a merging rule is referred to via the merge result, declared by the *into clause*. The "Classes" rule creates a new instance of the Class metaclass, carries out all mergings declared in its parent (ModelElements), and sets the feature list of the new class to be the union of all features from the left and right arguments.

There is a slight twist to the merging rule that takes the union of all features from the left and right model elements: the use of the *equivalent()* operator. This operator returns the *equivalent of the model elements to which it is applied* in the target model. The equivalent of an element is the result of a merge rule if the element has a matching element in the opposite model; otherwise it is the result of a transform rule. In short, this operator is necessary because the target and source metamodels may differ. This operator ensures that model elements from the source metamodel are expressed in the target metamodel before revealing the result of the composition.

Additional details on EML, including its merging strategies, and its support for different metamodels (e.g., MOF [1], EMF) can be found at [7].

3 Application of Model Merging

We now describe an application of model merging for managing model versions. The aim of this application was to develop MDA [2] support for an established and intensively used approach to software development process without changing it. This work was done as a joint effort between Fraunhofer FOKUS and an industrial partner. The focus here is to contrast previously gathered experience on managing model versions *without* the use of automated model merging, with a model merging-based approach, making use of EML.

In the following sections a short overview of the specific software development process will be given, and an example will be used to illustrate the approach used for merging model versions. This example will comprise only a fragment of the overall process but will be sufficient to illustrate the complete model merging approach used. The approaches to managing model versions will then be described: first, a semi-automated model difference analysis and merge facility will be described; and then our attempt at using EML rules to obtain similar results will be presented.

The first (semi-automated) approach has tool support; this model difference analysis and merge facility, implemented and used in an industrial project, is predominantly confidential. Presentations about the tool have been made [13]. For reasons of confidentiality, in this paper we describe a separate implementation of a similar merge approach in the Eclipse/EMF environment.

3.1 Context: The Software Development Process

We now describe the software development process that was to be extended and further developed to support MDA. The original software construction process supports all phases from planning through design and realization of the construction of enterprise information systems in an integrated environment. In the first phase the business rules and their relations are specified. The second phase allows the definition of Use Cases and their interrelations. In the third phase the flow of screens has to be specified, i.e., anticipated screens and the flow of control from screen to screen. This has to been done for each screen flow / use case; this description of screen flow can be seen as a behavioural description for the use cases.

For each screen flow, diagrams are derived in the fourth phase, which describe system boundary objects to the user, server control objects and the data entities linked to the screens. This derivation is done using transformations starting with the objects from phase three. The resulting diagrams will be manually augmented increasing the description of control and data flow in the system.

After this step a transformation can be used to generate a detailed class design for each of the diagrams received in phase four. These are still on a design level and not code, but are comparable to the code generation which is done in the next step.

All objects created or modified within the different phases are stored in a MOF repository. Passing from phase to phase is supported by transformations. Relationships between objects that are derived from other objects by these transformations are kept as traces within the repository.

To limit the degree of detail in this paper, we will restrict our example to the use case phase and design of screen flow section since the requirements for versioning and model merge are similar for all phases.

3.2 The Model Difference Analysis and Merge Facility

We now briefly describe the initial approach to providing model merge and differencing tool support for the development process presented in Section 3.1. We go into sufficient detail so as to be able to precisely understand what needs to be implemented using EML rules.

The existing tool offers team support facilities. It allows users to model different system aspects in parallel, to store them in model versions, and consolidate the results of parallel development. Consolidation is complex and time consuming. Within this environment a semi-automated approach is used. The person utilizing the diff/merge facility (model integrator) controls which model artefacts get transferred from the source to the target or are to be removed from the target.

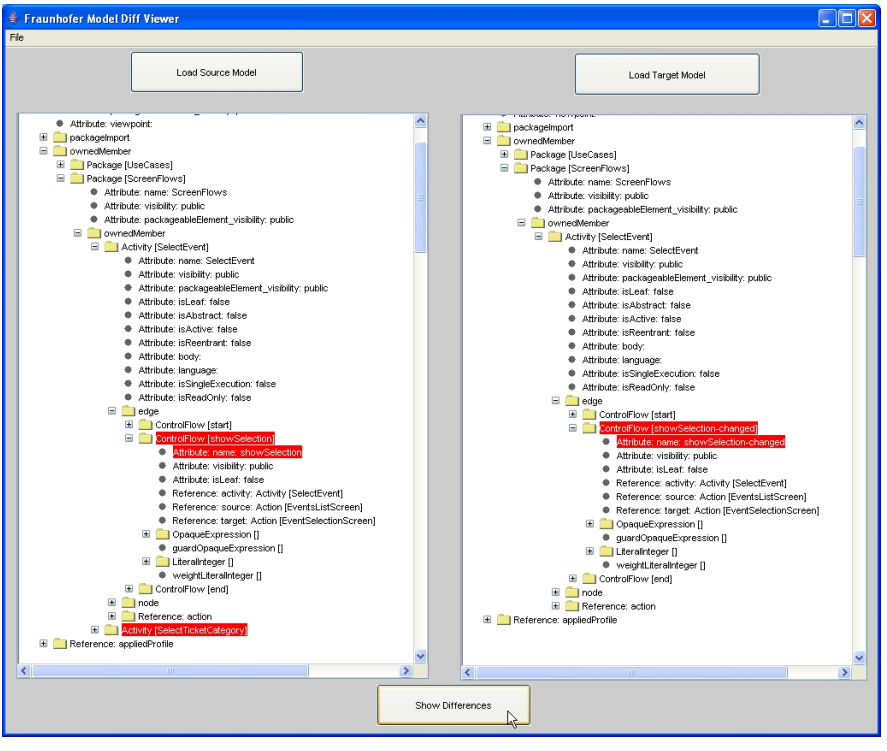

Fig. 3. The Model Diff Facility

The diff/merge process consists of three phases:

- Load the two versions of the model to be merged and determine the differences (the two versions are displayed in two sub windows).

- Navigate the artefact differences in the source and target window and decide whether to copy them from source to the target or delete them within the target window. In this phase this is done by only marking the artefacts as "to be" copied or deleted.
- After the above decisions have been made the resulting model is to be saved as a new version. This is the actual execution of the merge process.

3.2.1 Phase 1: Loading the Versions and Identifying the Differences

Only versions of complete models, and not parts of models, are supported. So the two versions of a model must have had a common subset of artefacts - at least the root element - before they evolved into separate versions.

The development environment is based on a MOF Repository [1]. Every MOF object obtains, at the time it is created, a unique identifier within the model which cannot be changed for the lifetime of the object. This MOF Id can be used to definitively establish an object's identity.

> *Instance typed objects have full object identity; that is, it is always possible to reliably distinguish one instance (object) from another. Object identity is intrinsic and permanent, and is not dependent on other properties such as attribute values. [from MOF 1.4 specification formal/02-04-03]*

The first step during loading therefore uses those modelling artefact identities to identify elements that have been dropped or created in between the creation of those two versions. Objects with MOF Ids in the older version that do not appear in the newer one have been dropped, those that appear in the newer but not the older version have been created. Changes will be marked and highlighted in the Diff/Merge browser (Figure 3).

Having two modelling artefacts representing the same object (i.e., with the same MOF Id) does not mean that they cannot be different. Their attributes or references may have changed, e.g., new ones may have been added, deleted or changed. We therefore have to elaborate how to discover a difference in more detail.

To determine differences, we can make use of the reflective module of MOF. This has the advantage that the difference algorithm can be used for other metamodels too without changes. The reflective module allows working on the model and metamodel level without using metamodel specific interfaces that means without having knowledge of the metamodel in advance. For example, we can start with a modelling object, find its attribute names and values, its references, multiplicities of attributes and references and so on.

For each pair of modelling artefacts (i.e., a new version and an old version) with the same MOF Id we

- Check whether the attributes (names and values respecting the multiplicity) are different.

  For primitive types we directly compare the values. For complex types we have to recursively compare the objects pointed to for differences.
- Check whether the references (referenced objects) are different. This is also done recursively.

If we have found differences, we mark them and also mark the objects we started with as different. We have to be aware of cycles during the recursive search. If we are not interested in highlighting all differences in the browser, we could abort the search for differences after the first difference found for an object. To identify all changes we have to look from version A to version B for differences and vice versa.

As a result we will have a list of all modelling artefacts in version A that have changed or added to version B and all those in version B which have changed or are added to version A. This holds for all attributes and references in the model artefacts.

This reflective approach has the advantage that we do not need to know anything specific about the metamodel. Its disadvantage is that we perhaps compare too many objects that haven't changed. If we have knowledge about the metamodel and more specifics about the way changes could have taken place within the model, we could restrict the number of objects we have to look at. For example, only modelling artefacts that are visible in the modellers GUI could have been changed by the modeller.

3.2.2 Interpreting the Differences and Identifying the Elements to Merge

Within the merge phase we have a directed operation, i.e., one specific version of the model is the source and the other the target. The source elements may be added/merged from the source to the target but not vice versa. Elements may be deleted from the target that are not present in the source. This phase is normally done with support by a human being, who decides which differences should take part in the merge in which way and marks them as to be added from the source to the target or deleted from the target etc. The following kinds of objects can be identified:

- Objects (identified by their MOF Id) that are present in the source but not in the target may be added to the target (marked as *add*).
- Objects (identified by their MOF Id) that are present in the target but not in the source may be deleted from the target (marked as *delete*).
- Attributes present in a modelling artefact in the source but not in the corresponding target can be created in the corresponding target artefact (mark as *add*).
- Attributes present in a modelling artefact in the target but not in the corresponding source can be created in the corresponding target artefact (mark as *delete*).
- References or links between objects to be added or deleted
- Changes in modelling artefacts that are present in both versions but which have changed their values

3.2.3 Merge Phase

This phase executes the identifications and decisions made in phase 2 and creates a new target model in the repository. It takes the following steps:

1. Delete all references to objects that are marked to be deleted
2. Delete all the objects marked to be deleted
3. Add all objects marked to be added (without their attributes and references, because they may reference objects not present)

4. Add all attributes to be added
5. Add all references to be added

Since it is not guaranteed that the decisions made in phase 2 lead to a consistent metamodel instance, we have to make a constraint check to validate the resulting model.

3.2.4 The Metamodel

The metamodel used within the industrial project is proprietary. Within the OMG presentation [13] a single slide gives an overview of the package structure used within the metamodel. This structure is shown in Figure 4.

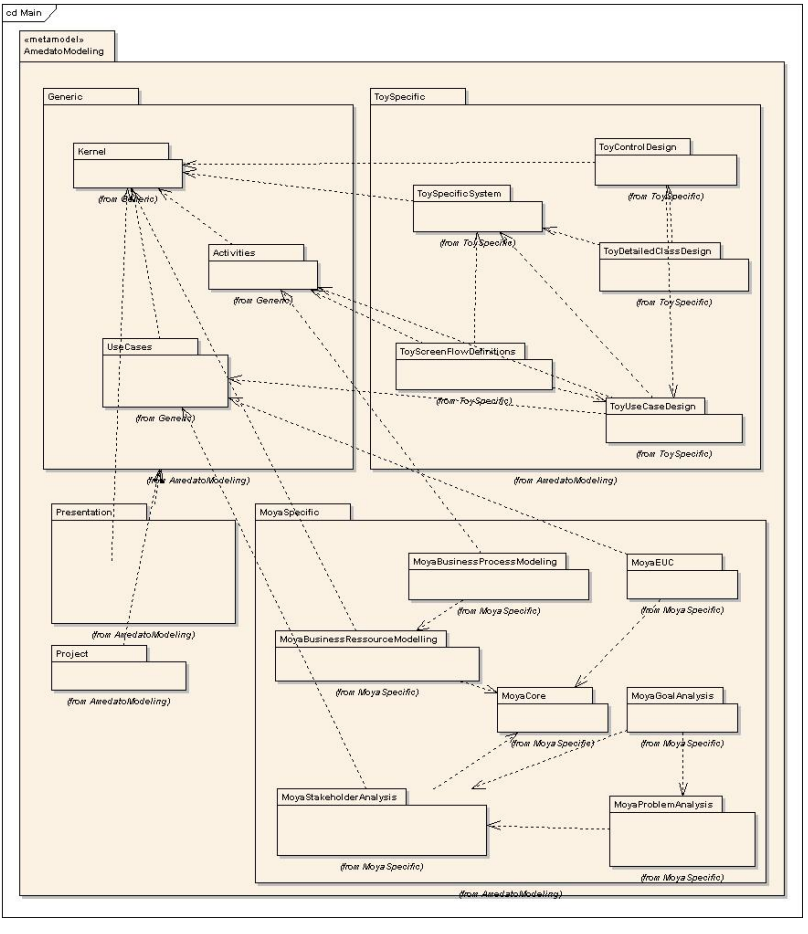

Fig. 4. The Metamodel Package Structure

The metamodel is based on parts of UML 2.0, specifically UseCases, Activities and Kernel elements. These elements are specialized for their usage in the different development phases in the specific packages. These specific modelling elements are

offered through a graphical modelling tool to the user. The models are stored within a MOF repository generated from the metamodel. Additional metamodel packages are defined for project metadata and the graphical representation data of the model. Within our examples we will concentrate on the use case screen flows.

For use case design the modeller will offer the following (specific) elements: actors, use cases, association, generalization, include and extend dependencies. At model creation time the user has to enter an identifier (e.g. for use cases) that will be unique within the model. These identifiers are not as secure as the MOF ids assigned automatically, but can also be used to identify objects in the merge process.

For the specification of screen flows we will have a screen (derived from activity), transitions between screens, initial and final nodes. These elements will also have specific identifiers assigned by the user during creation.

3.3 Model Merge with the Epsilon Merging Language

In this section we briefly outline how we use EML to manipulate the models, identify matches, and carry out the kinds of merges discussed in the preceding. In this sense, we are demonstrating the value and applicability of generic model composition technology for helping to solve a specific model management task, namely model differencing and merging of different model versions.

With EML, the developer defines a set of rules which will be applied to two metamodel instances (one, for example, called the *left* instance, and the other the *right*). Applying the rules will partition the modelling elements into the four categories noted earlier, in Section 2.2.1, i.e., elements that do not match, elements that match and conform, etc. This partitioning of model elements will be used to separate the newly added, deleted and changed from the unchanged elements within the two versions.

For the matching model elements, and hence writing the matching rules in EML, a combination of attributes/elements will be used that have the capability of uniquely identifying the element. For example, for use cases, the use case name suffices, and for screen flows the activity name will suffice. For associations we need a more complex construct of source and target, e.g., their role names, to identify it. A better approach would be to use the MOF identifier or the corresponding EMF identifier and a comparison function that checks two objects for identity, as we did with the approach previously described. Support for use of MOF identifiers was not available in the version of EML trialed for this experiment, but has since been added. In the case that a modelling tool preserves unique MOF identifies, an identity-based *matching strategy* (currently implemented for both MOF and EMF) can be used to identify matches based solely on element identity. We use EML conformance rules to identify changes in other characteristics of an element (e.g. changes in an element's visibility, multiplicity, direction for navigation, etc.). Applying the matching to two model versions will lead to the partitioning mentioned earlier. We discuss the information provided in each case with respect to the problem at hand.

Category 4 elements are a hint that we have forgotten to define a match rule needed for the model versions. In this case we infer that our rule set is incomplete and must augment it.

Category 1 elements are those that are contained in both model versions. However, the partitioning and inclusion of elements in this category does not say anything about them being different since contained elements like attributes or references may have changed. Elements of category 1 match and conform with their opposites. This means that even if some attribute values are different, the designer of the EML specification has decided that they are not important enough to raise a conformance issue – then they would belong to category 2.

Category 3 elements are those elements that lead to changes. These may be elements that have been deleted or added in one model version.

Elements of Category 2 have a corresponding partner but an expression that was checked in the conformance part of an EML rule has changed (e.g., the type of the attribute, the visibility of the element, direction of an association etc.).

A sample of the EML matching rules used for such an application is presented in Figure 5. We present an example of a merge rule thereafter.

```
abstract rule Elements
     match l : V1!Element
     with r : V2!Element {

     compare { return l.owner.matches(r.owner); }
}

abstract rule NamedElements
     match l : V1!NamedElement
     with r : V2!NamedElement
     extends Elements {

     compare { return l.name = r.name; }
}


rule UseCases %% matches only on names
     match l : V1!UseCase
     with r : V2!UseCase
     extends NamedElements {
}

rule Properties
     match l : V1!Property
     with r : V2!Property
     extends NamedElements {

     compare {
       return l.type.matches(r.type) and
        l.association.matches(r.association);
     }
}
```

Fig. 5. Example EML match rules to support model version management

The manual decision process used in the earlier approach, where a person responsible for the merge decides which elements should be added or deleted within or to the target model, cannot be fully used with EML since the merge and

transformation rules will applied automatically. However, such a process could be supported with EML if *traceability information* (e.g., sources of elements in a merged model) can be recorded. [15] presents an approach to recording traceability information in EML. In general, if it is desired to be able to apply a manual decision-making process in concert with automated merging tools, a less strict set of rules should be written (e.g., leading to more elements in categories 2 and 4), which should lead to more exceptions during the automated merging process (in the case of increased elements in category 2) or more diagnostics suggesting model elements to which no rule applied (in the case of increased elements in category 4). Within match blocks an exception handling-like mechanism can be used. In practice, we attach a *fix* block to a *compare* rule; these fix blocks can be executed during a matching process. Their purpose is to allow modular conflict resolution and reporting. To add support for manual decision making we intend to split the matching/merging process into two distinct processes. The output of the match process will be a model (trace model) that will store the identified match-relationships (and the category in which each belongs) thus giving the opportunity to revise/correct/enhance the trace model before feeding it back into the merging process.

The merging process for model versions can be done (in the simplest case) automatically; however, this depends on the policy followed for team coordination and synchronisation. This in turn will dictate whether we will have more or less conflicts during the merge. A merge rule for this process is shown in Figure 6.

```
auto rule ModelWithModel
  merge l : V1!Model
 with r : V2!Model
 into m : Vm!Model {

  if (l.name <> r.name){
        m.name := l.name + ' merged with ' + r.name;
  }
}
```

Fig. 6. Example EML Merge Rule

3.4 Comparison of Both Approaches

The MOF reflective approach described in Sections 3.1-3.2 is independent of the metamodel; EML is also metamodel independent. In general, in both approaches a person is needed to resolve the conflicts appearing during the merge and to determine the merge strategy. The merging afterwards is done automatically. With EML this may require some experimentation with the matching and merging rules in order to determine where and when human feedback can best be injected. We found this not difficult to do with EML, particularly because the rule structure inherent in any EML program is helpful in determining exactly what kind of feedback is needed from a human operator due to a failed rule. Including traceability information, as in [15], would also help with this process.

For complex metamodels like the UML 2.0 metamodel the efforts for obtaining a complete set of matching rules needed for the EML approach will not be negligible; for this reason, the developers of EML are exploring matching strategies, e.g., based

on the ideal of a weaving model. With the strategies currently implemented within EML, this is straightforward. Another issue is dealing with the frame of rules, i.e., their domain of applicability. Frames for the rules may be generated from the metamodel using an Epsilon helper utility [5].

In general, MOF identifiers may not be available for determining the identity of objects since some tools do not persist such identifiers (e.g., Rational Studio Architect). While this is not a problem that a model composition tool such as EML can overcome, the user of automated composition tools must be aware of these limitations, as they will impact on the human intervention needed to deal with failed composition rules.

With EML, it is currently not possible to support the conflict decision process as it currently exists. The process of deciding how to resolve conflicts must be done in advance and coded within the merge and transform rules set. The whole process afterwards can be executed predominantly automatically, but human intervention will be needed to deal with conflicts that have been overlooked.

4 Conclusions

We have discussed the problem of model versioning, and explored its use in an industrial software development process in which a human-intensive approach was used to identify and reconcile differences. This approach was compared with that offered by an automated model composition solution based on EML. The automated approach was found to be appropriate for dealing with many of the problems of model differencing and model version management, though it is by its nature incomplete. A particular challenge with using an automated rule-based solution like EML for model versioning will arise in terms of producing a complete set of rules for large metamodels. EML supports a notion of a *merging strategy* which provides a default set of rules for compositions involving models from the same metamodel, and this may help in providing default rules for comparison and differencing for managing model versions. There are also issues in terms of using a model composition tool with other modelling tools, e.g., for maintaining visual presentations.

By using EML to merge model versions we identified two key limitations: limited support for exception handling and support for managing MOF identifiers. Support for both features has been added to EML as a result of this experiment.. Technical limitations (e.g., the need to support additional modelling technologies, and the utility of a separate model comparison language) have also been identified, and this will guide our future work. We also plan an additional experiment to demonstrate that EML is sufficient to carry out all key tasks of model versioning, as identified in Section 2.

Acknowledgement

The work in this paper was supported by the European Commission via the MODELWARE project. The MODELWARE project is co-funded by the European Commission under the ”Information Society Technologies” Sixth Framework

Programme (2002-2006). Information included in this document reflects only the authors' views. The European Commission is not liable for any use that may be made of the information contained herein.

References

1. Object Management Group. Meta Object Facility official web-site. Internet resource. http://www.omg.org/mof/.
2. Object Management Group. Model Driven Architecture official web-site. Internet resource. http://www.omg.org/mda/.
3. Object Management Group. XMI specification. Internet resource. http://www.omg.org/technology/documents/formal/xmi.htm.
4. Modelware IST Project. Internet resource. http://www.modelware-ist.org.
5. D. Kolovos, R.F. Paige, and F.A.C. Polack. The Epsilon Object Language (EOL), to appear in *Proc. EC-MDA 2006*, LNCS, Springer-Verlag, July 2006.
6. Atlas Model Management Architecture, available at http://www.sciences.univ-nantes.fr/lina/atl/AMMAROOT/, last accessed February 2006.
7. Epsilon Model Management Platform, available at http://www.cs.york.ac.uk/~dkolovos/epsilon, last accessed February 2006.
8. Enterprise Architect. http://www.sparksystems.com.au, last accessed January 2006.
9. D. Ohst, M. Welle and U. Kelter. Differences between Versions of UML Diagrams. In *9th European Software Engineering Conference*, pages 227–236. ACM Press, 2003.
10. Y. Lin, J. Zhang, and J. Gray. Model Comparison: A Key Challenge for Transformation Testing and Version Control in Model Driven Software Development. In *OOPSLA Workshop on Best Practices for Model-Driven Software Development*, 2004.
11. M. Alanen and I. Porres. Version Control of Software Models In *Advances in UML and XML-Based Software Evolution*, Idea Group Publishing, 2005.
12. M. Alanen and I. Porres. Difference and Union of Models. In *Proc. UML 2003 - The Unified Modeling Language*, LNCS 2863: 2-17, Springer-Verlag, Oct. 2003.
13. O. Kath, The AMEDATO Solution - A Success Story For Model Driven Technologies, Burlingame, CA, U.S.A. December 5-9, 2005 (AMEDATO Presentation: mda-user/05-12-01)
14. Revision Control Wikipedia Entry, http://en.wikipedia.org/wiki/Revision_control, last accessed February 2006.
15. D.S. Kolovos, R.F. Paige, and F.A.C. Polack. On-Demand Merging of Traceability Links with Models, submitted April 2006. http://www.cs.york.ac.uk/~dkolovos.